MLPE: An Extensible Multi-Language Programming Environment

Louis-Julien Guillemette

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2004

Your file Votre référence
ISBN: 0-612-94740-8
Our file Notre référence
ISBN: 0-612-94740-8

ABSTRACT

MLPE: An Extensible Multi-Language Programming Environment

Louis-Julien Guillemette

Sizable projects often involve the combination of multiple languages, and many projects involve the use of one or more languages designed specifically for the application. There exists a primary need, on the one hand, to facilitate the cooperation of several languages, and on the other hand, to facilitate the implementation of new ones. Correspondingly, one can identify two kinds of software: (1) mixed languages systems, which are those systems combining parts written in several languages, and (2) language prototyping tools, which are tools meant to facilitate the construction of prototype implementations of a programming language based on some formal description of it. This thesis reports an attempt at constructing a programming environment that combines the requirements of the two.

The thesis presents MLPE, a programming environment designed for generic support of programming languages (i.e. meant to host an open-ended set of languages.) Cross-language operability is enabled through a set of common programmatic abstractions, namely a common type system and module system. MLPE is extended to support three languages in common use, namely C, Java and Haskell. A language prototyping tool designed to complement the programming environment is also presented. The prototype implementation of a simple procedural language is constructed and put to use in the context of the programming environment.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my two supervisors, Dr. Patrice Chalin and Dr. Peter Grogono, for their support and encouragement. I thank Dr. Grogono for his careful advice. In many instances has he helped me renew my interest and get going with new ideas. I thank Dr. Chalin for his advice as well. His elaborate suggestions and criticism have been invaluable help.

Table of Contents

Chapter 1 Introduction

The diversity of programming paradigms and languages in use today is a basic fact of software engineering. Sizable projects often involve the combination of multiple languages, and many projects involve the use of one or more languages designed specifically for the application. There exists a primary need, on the one hand, to facilitate the cooperation of several languages, and on the other hand, to facilitate the implementation of new ones. Correspondingly, one can identify two kinds of software:

1. mixed languages systems, which are those systems combining parts written in several languages, and
2. language prototyping tools, which are tools meant to facilitate the construction of prototype implementations of a programming language based on some formal description of it.

This thesis reports on an attempt at constructing a programming environment that combines the requirements of these two kinds of software.

1.1 Mixed-Language Systems

When a module invokes a service of a module written in another language through some sort of foreign language interface, we are in presence of a mixed language system. This is the case, for instance, when a Java method invokes a C function through the Java Native Interface, or JNI [Liang99]. Such a foreign language interface defines conventions for accessing some entities belonging to the foreign language; it typically defines functions for constructing values, extracting component values, calling functions or object methods, and so forth.

A foreign language interface provides interoperability for a pair of languages. In the general case, one may consider interoperability across a set of languages. For instance, this is what CORBA [CORBA] does for object-oriented languages: by offering a standard way to define service interfaces implemented in any object-oriented language, it allows modules written in various languages to interact.

This thesis aims at constructing a system that provides basic interoperability among languages based on different paradigms, namely imperative, object-oriented and functional languages.

1.2 Language Prototyping Tools

One significant application of formal semantics is to rapid prototyping – the synthesis of an implementation for a newly defined language based on its semantic description [Schmidt95]. Various styles of semantics have been used in this way and an impressive body of literature has accumulated on the topic. An overview is provided next.

The most common example of a semantic method that can be used for rapid prototyping is *denotational semantics*. In denotational semantics, the meaning of a language is defined as a set of functions mapping syntactic constructs into semantic domains. One way to obtain an implementation of a language is to encode its denotational description (i.e. functions and domains) using a functional language – once compiled, the encoded description provides an interpreter. SIS [Mosses76] and PSI [NN88] are some early systems that processed denotational semantics to synthesize a language implementation.

Formal semantics is an especially difficult subject as it encompasses the challenges of expressing and carrying out computations. Much theoretical work has been done on semantics since the 60s, with the most significant advances being attributed to Scott and Strachey [Stoy77].

Even though denotational semantics has gone a long way to make difficult computational concepts more precise, it bears problems of its own. Reasons why denotational semantics has not become a common engineering tool despite its aptitude and soundness are known (see e.g. [Mosses94]). The problems are mostly of a pragmatic nature: denotational descriptions are hard to compose and read, they are not modular, and it takes a considerable mathematical background for readers to understand them.

Several attempts were made at devising semantics with better pragmatics, either as refinements of denotational semantics or starting on different ground. To state just two:

1. Modular Monadic Semantics [Moggi90] uses the concept of a monad (borrowed from category theory) to make denotational descriptions modular, in the sense that

individual language features (such as the calling mechanisms or the use of assignment or references) can be changed independently of other features. Monads bring complications of their own, however: they provide modularity, but they are themselves hard to combine, and general ways for combining monads had to be investigated later [Espinosa95, LHJ95].

2. Action Semantics [Mosses92] is a semantic framework based on completely different abstractions. The central notion is that of an "action", which is a computational entity that can be applied, or "enacted", to produce a result. Actions have a number of so-called "facets" to represent the different aspects of a construct: a functional facet to record the values produced by a construct, a declarative facet to track its declarations, and an imperative facet to track its side effects. Action semantics defines primitive actions along with a set of combinators that are used to assemble semantic descriptions. The fact that individual actions affect a specific set of facets and leave the others facets unaffected makes action semantics modular. Also, the notation for actions semantics being closer to English than, say, denotational semantics, makes the formal descriptions more readable. ACTRESS [BMW92] and OASIS [Ørbæk94] are two systems that synthesize a compiler given an action semantics description of a language.

Despite the many efforts at creating a semantics that is more suitable for general use than classical denotational semantics, no semantics has yet gained widespread popularity among programmers [Schmidt97]. In the current state of affairs, work in formal semantics (or rapid prototyping) requires a high degree of specialization.

The system developed in this thesis is not specialized toward a particular style of semantics. It rather aims to facilitate experiments with a suitable formalism, but does not itself provide functionality for processing semantic descriptions.

1.3 Thesis Outline

The thesis reports the construction of an experimental programming environment, named MLPE, designed for generic support of programming languages (i.e. meant to host an open-ended set of languages) and that of a language prototyping tool. MLPE is extended

for supporting various languages. Finally, the use of a prototype language implementation is tackled in the context of the programming environment.

The goals of the project are clarified in Chapter 2, where the desirable properties of the programming environment and prototyping tool are stated. The detailed design of the programming environment is presented in Chapter 3. Chapter 4 shows how MLPE is extended to support three languages in common use – namely C, Java and Haskell. The design of to the language prototyping tool is presented in Chapter 5. In Chapter 6, the prototyping tool is used to implement a simple imperative language, and the resulting implementation is integrated into MLPE. A sample project is developed using MLPE in Chapter 7. An overview of related work in provided in Chapter 8. Chapter 9 summarizes the work and indicates directions for future extension.

Chapter 2 Project Goals and Project Outline

This chapter states the desirable properties of MLPE and the language prototyping tool. Some early issues that pertain to the entire system are addressed – mainly with respect to the implementation language we use. Finally, an overview of the organization of MLPE is given.

2.1 Desirable Properties of the Programming Environment

The programming environment is meant to host support for various languages and provide mechanisms for modules written in these languages to interoperate.

In order to be realistic, MLPE has to cope with languages in common use. Therefore, MLPE must have provisions for implementing support for languages using existing implementations (say, compilers) as well as semantic-based ones. The foremost requirement is *extensibility*, in two respects: it should be easy to implement support in MLPE for

- a language whose implementation is given (e.g. Java given the JDK), and
- a simple language whose semantics is given.

For this purpose, the system should define an abstraction for an *implementation* of a programming language – it is intended that this abstraction be realized for each supported language. The abstraction should be sufficiently generic so that it can accommodate implementations of different nature (e.g. an existing compiler vs. an implementation constructed using the prototyping tool.)

In order to allow interoperation of the supported languages, MLPE should provide a set of abstractions that can be used uniformly across all the supported languages. This set of abstractions should be captured by a *type system*; this type system shall be defined independently of any particular language's type system, and is meant to provide a common semantics for the languages to interoperate. The set of abstractions should be

- sufficiently rich to allow effective interoperation, yet
- sufficiently generic to be used in languages of different paradigms.

The remaining requirements are pragmatic in nature and are meant to ensure that the system can be useful in practice.

MLPE should be an *interactive* programming environment, where modules written in the supported language can be loaded dynamically, and the user can invoke their services interactively.

Moreover, MLPE should be an *integrated* environment, in the sense that the

- various language implementations should be registered in the programming environment and invoked automatically by the system, and
- modules of an application developed using the system should be managed automatically by the system.

To clarify, the second point involves "building" the modules (i.e. invoking the compilers as needed to produce any intermediate files) and loading them automatically when their source files have changed.

As a side requirement, it is deemed desirable that the response time for the operation of building and loading the modules be low, so as to permit a tight code-and-test cycle.

2.2 Desirable Properties of the Language Prototyping Tool

The prototyping tool is meant to facilitate the construction of a programming language interpreter based on some formal description of the language.

As stated in the Introduction, MLPE is not designed with a particular formalism in mind for describing the semantics of a language. It is intended that the user should be able to use any suitable formalism. Thus the primary requirement is *expressivity*: it should be convenient to express the semantics of a language using any formalism that is suitable for implementation. In order to meet this requirement, MLPE relies largely on its own implementation language, **Haskell** (to be discussed in Section 2.3.1). It is intended that the user should either express the semantics using Haskell functions (as would be the case with a denotational description of a language), or express the semantic description as data, using custom data types, and provide support functions for tracing the semantics (as would be the case with e.g. action semantics or operational semantics.)

Regardless of the chosen semantic method, there is a need to express the syntax of the language. MLPE should provide a convenient facility for expressing the syntax of a language and for integrating the syntactic processing with a semantic processor implemented in Haskell (to be developed separately by the user). In order to make it convenient to describe syntax, the system should provide a tool that is

1. fully declarative: the user should be able to describe the syntactic features of a language in a declarative manner.

In order to facilitate the integration of syntactic processing with semantic processing, the tool should provide an interface that is

2. accessible through an interface in Haskell. The user should be able to encode the syntactic description data in Haskell, and to feed this data to the system in order to obtain a syntactic processor.

It is intended that a complete prototype implementation of a language be constructed by expressing both the syntax and semantics of the language within one or more Haskell modules.

2.3 Overview of MLPE

This section provides an overview of MLPE and is meant to serve as a prelude to the detailed description of its components in subsequent chapters. For a direct appreciation of the system, the reader may prefer to read Chapter 7 first, where a demonstration of MLPE is given.

This section contains a justification of Haskell as the choice of implementation language (Haskell) along with a description of the overall design of MLPE.

2.3.1 Choice of Language and Implementation

A significant decision to make early on is the choice of implementation language. It is relevant to consider the nature of work to be carried out using the language; the most involved implementations tasks are:

- the modeling of formal systems, such as the syntax and semantics of programming languages, and

- systems programming, mainly for interfacing foreign runtime systems when using existing language implementations.

The primary reason for choosing Haskell is that its higher expressivity facilitates modeling of formal systems. Its elaborate type system proves very handy for expressing formal semantics. In addition, its quality of purely functional language allows the precise identification of those regions of the code which have side effects (that is, functions whose types involve IO's.) This is useful for differentiating between the parts that are to be modeled using just algebraic data types and pure functions (for instance, we may want to model formal semantics in this way) from the side-effect-intensive code fragments such as those parts which involve system programming. Also, the static type checking of Haskell provides an additional degree of safety by ruling out run-time type errors.

On the other hand, Haskell makes system programming more challenging. The language does provide an interface to C (the FFI) which provides basic connectivity. However, we need additional services that are not defined in the language. For instance, the Haskell standard does not define a way to dynamically load a Haskell module—feature needed to support Haskell in the interactive environment. For such non-standard features, we have to rely on a *particular* Haskell implementation. The designated implementation is the Glasgow Haskell Compiler (GHC), which provides an elaborate library (the so-called Hierarchical Libraries) that addresses many of these issues. GHC is being very actively developed and also has the advantage that it produces the most efficient code compared to other available implementations.

2.3.2 Design Overview

MLPE is designed to host implementations of various languages, while its language prototyping tool provides a facility for implementing support for a particular language (as shown in Figure 1.) The core of MLPE and the language prototyping tool are not coupled, however (in the sense that the programming environment does not contain explicit references to the prototyping tool, and vice-versa.)

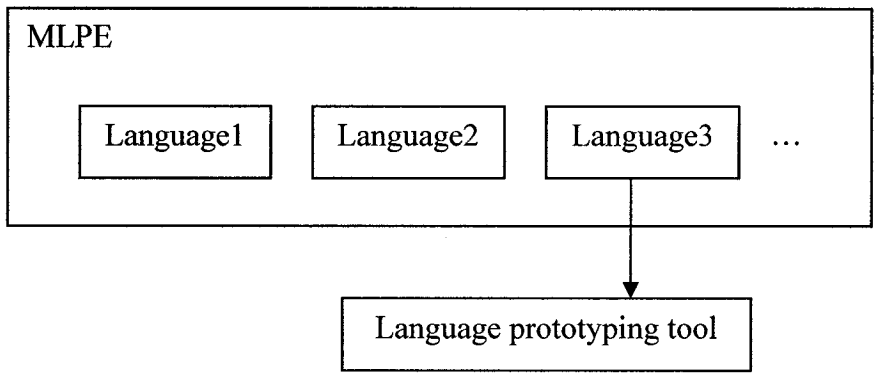

Figure 1. Relationship between the programming environment and the language prototyping tool.

The programming environment is relatively complex as it involves several inter-related concepts. It is based on two programmatic notions:

- a common type system, which provides common semantics for defining and accessing services uniformly across all the supported languages, and
- a common module system, to allow modules to be defined in any of the supported languages; such modules can export a number of services for other modules to use, and access modules exported by other modules; these services are defined using the common type system.

The third important notion is that of a language implementation. In the scope of MLPE, a language implementation is an entity that provides mechanisms for

- accessing the common type system from within the object language,
- defining modules using the object language, and
- loading such modules in the programming environment.

For the purpose of accommodating existing implementations of programming languages, MLPE provides an interface for connecting compilers into the environment. MLPE provides a facility for the user to specify the intermediate files that would be created by such compilers in a particular application. It manages the invocation of the compilers based on these specifications, much in the manner of a traditional 'make' tool [Make].

Finally, MLPE provides an interactive (textual) interface that allows the programmer to load modules into the environment and invoke their services.

The language prototyping tool follows a simpler design. It provides a facility for defining the syntax of a language, while the semantics should be specified (and implemented) by any suitable means in Haskell. The facility for defining syntax consists of a set of data types for specifying the syntax of the language, along with a set of support functions that construct analysis functions (i.e. a scanner and parser) given such specifications. It is intended that the user write down the specification of the syntax of the object language as *values* in these types as part of the implementation of the language, and invoke the functions of the prototyping tool to obtain an implementation (i.e. scanning and parsing functions).

Chapter 3 The Mixed-Language Programming Environment

This chapter presents the core component of MLPE – the programming environment itself. It begins with an overview of its design and then covers the detailed design of its components.

3.1 Design Overview

The design of MLPE is elaborated over five main elements, the first of which is the most significant:

1. A set of programmatic abstractions that enable cross-language operability. The system defines a number of abstractions that have a uniform meaning across all of the supported languages. There are two components to this:
   a. A *type system*: MLPE defines a type system that allows the programmer to define and access program entities uniformly across all supported languages. The type system is primarily made of primitive data types, structured data (pairs and lists) and functions. This type system thus provides a basic mechanism to pass data around and define functionality.
   b. A *module system*: MLPE allows the programmer to implement program modules in any of the supported languages. Modules are meant to be dynamically loaded into the environment. They define a scheme for implementing program units that can provide certain services, as well as a means for these units to make use of services defined in other units (that is, to import services from other modules.) The services defined in a module are defined as values in the common type system.

The foremost feature of the design is that it is centralized: all interactions between modules are actually mediated by the common run-time system (at run time.) In this way, only interactions between individual modules and the runtime system need be considered.

The basic strategy for realizing the common abstractions in a given target language is the following:

- For programming in the common type system, a representation of the values and functions in the common type system in the target language is defined. The implementation problem, then, is to define a way for the run-time system to import the representation from the target language, and to export its own internal representation to the target language. Whenever a value is passed from one module to another, it is in fact imported into the common run-time system and exported to the other language.

- For implementing modules in the target language, MLPE provides the target language with an interface for defining a module, and an interface for a module to access objects from the modules it imports. These interfaces need only be in terms of the target language's representation of the common type system – it is the runtime system's responsibility to import and export the values as needed. The modules can then be programmed in terms of their own language's representation of the common type system.

For the purpose of interfacing between MLPE and the target language, the above services (for accessing the type system, defining a module allowing a module to access services of modules) are provided in the form of either a library (as is the case with C, Java and Haskell in Chapter Chapter 4) or special syntax (as is the case with the prototype implementation in Chapter Chapter 6.) This interfacing is illustrated in Figure 2.

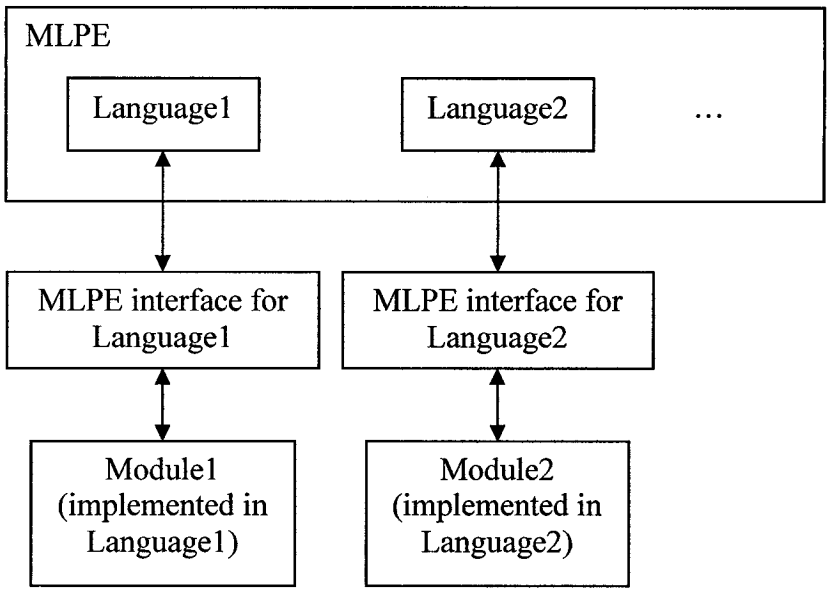

Figure 2. Centralized design of MLPE.

The other four design elements constitute the programming environment itself, whose main objective is to load modules and provide the run-time support for these modules to co-operate. In brief, they are as follows:

2. A *project specification facility*: MLPE provides a means to specify a *project* made of several modules. In a project specification, the source files of each module are identified, as well as any intermediate files that are needed in order to load the module. We call these intermediate files *resources*.
3. A set of *interfaces for language processors*: These provide an interface for connecting a language implementation into MLPE.
4. A *make tool*, which makes use of the language implementations to generate the resources of a project so as to prepare its modules for loading.
5. A *shell*, where the user of the environment can initiate operations over the project (such as building the project and loading its modules), inspect the contents of modules, and invocate services provided by the modules. The loader also manages the loading of modules.

These four components are depicted in Figure 3.

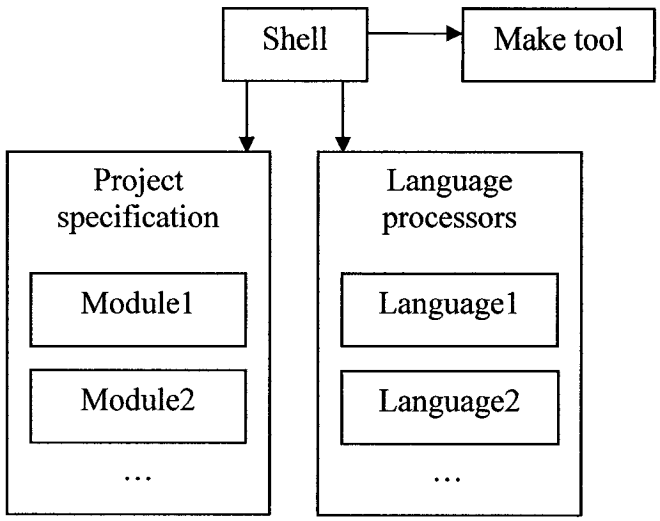

Figure 3. Main components of MLPE.

The programming environment is abstract (and useless) until it is populated with actual implementations of particular languages. To implement support for a language in MLPE, one realizes the predefined interfaces of the language processors using an arbitrary implementation of the language.

Project Specifications

The first step in writing a project is to write its specification. This specification defines the concrete, or static constituents of the project. It lists the modules themselves, as well as their source files and any intermediate files (resources) needed to load the modules (as shown in Figure 4.)

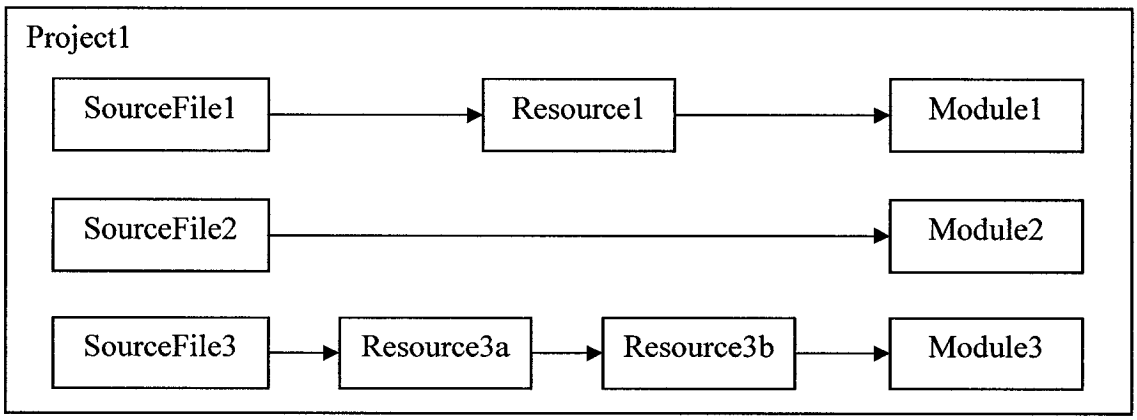

Figure 4. The constituents of a project.

To take a concrete example, consider how support for C could be implemented. The source file of a C module would normally be compiled into object files and, in order to be able to dynamically load the module, the object file would be linked into a dynamic library. In this case, the resources involved are the individual object files and the dynamic library. To account for the different nature of such resources, the specification assigns explicit *types* to resources (such a "C object file" or "dynamic library"). Similarly, each source file has a specified *source language*. MLPE uses this type information to determine which language processor to use for handling a given source file or resource (more on this follows in this overview.)

In addition to types, resources may have explicit dependencies among themselves. For instance, in the example above, the dynamic library depends on all of its object files. The explicit specification of these dependencies allows the make tool to initiate the generation of the individual resources in a suitable order.

Language Processors

MPLE provides three abstractions to represent language processors. The first two model processors that can generate individual resources when building a project:

1. compilers, which take source files to resources, and
2. linkers, which take resources to other resources

The third is the critical one:

3. loaders, which load a module from a specified source file or resource into MLPE.

Continuing with the example of a module in C, the processing of the constituents by the language processors is illustrated in Figure 5.

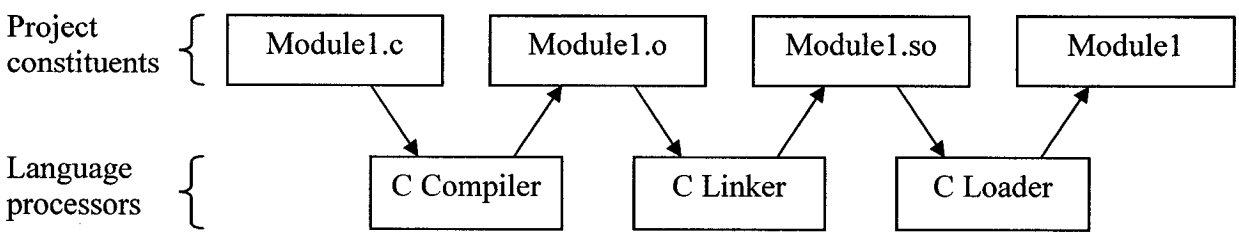

Figure 5. Processing of individual constituents of a project.

These three abstractions are defined as Haskell types, and particular language processors are defined as objects of these types.

In the definition of a language processor, the types of the resources it handles are explicitly specified: a compiler has a given source language and target resource type; a linker has a given source and target resource type; and a loader has a given input resource type or source language. For the purpose of building the project, this allows the make tool to select the appropriate processors for building a given resource. It also allows the shell to select a loader for loading a particular module, according to the type of the resource that is associated with that module.

The input to compilers, linkers and loaders are specifications of source files, resources and modules. When using a concrete implementation such as, say, GCC, to realize a compiler or a linker, the compiler or linker in question is no more that a simple wrapper around the concrete implementation – what it does is merely to invoke the concrete compiler or linker to build the specified resource.

Loaders are generally more complex than compilers and linkers, as they model the very central task of loading a module in a given target language into MLPE. A brief overview of the loading process is given in the following two paragraphs; the process is explained in more detail and illustrated in Section 3.5.3.

Based on some concrete representation of a module stored in a resource, a loader instantiates MLPE's internal representation of that module. The internal representation, defined by a Haskell type, contains a map of the objects defined in the modules along with a listing of the imported modules and some other components (see Section 3.3).

An important feature of a loader is that it provides a means to equip the loaded module with an interface through which it can invoke services from other modules. We call the interface in question an *import context*; it is a function that allows a loaded module to lookup symbols defined in any of the modules it imports. It is provided to the module in this way: the client of the loader (in our design: the shell) first invokes the loader and thus obtains the internal representation of the loaded module; along with this representation, the loader returns a callback function to the client – it is assumed that the client calls this

function back and passes as argument a valid import context. Then, it is the loader's responsibility to expose this lookup function to the loaded module.

The Make Tool

The make tool's duty is to initiate the generation of the various resources of a project. It does a job very similar to that of a conventional (Unix) make tool. Based on the dependencies listed in the resources' specifications, the make tool identifies a suitable order for generating the resources, and invokes the compiler and linkers to build them.

The Shell

The shell provides a user interface to MLPE. It is a simple textual interface that looks like a plain language interpreter, but it gives access to modules and objects defined in any of the supported languages. It has a simple language to take commands from the user and do evaluations and invocations over the loaded modules' objects. Programmatically, it is the controlling part: it holds the project specification and a collection of language processors, invokes the make tool for building the project, and instantiates the loading of the individual modules.

The loading of modules in the shell is an involved task. In order to identify which modules need to be loaded, the shell performs "module chasing" – that is, it recursively loads the imported modules. Once the modules are loaded, the shell needs to provide each module with its import context so that the modules can interact. The loader sends the import context back to the module's loader, which in turns exports this function to the module implementation, so that this module can access objects defined in its imported modules.

3.2 Common Type System

The purpose of the common type system is to allow modules written in diverse languages to exchange data and functionality – it provides common semantics that can be addressed uniformly across these languages. It is purposely small, so that it can be implemented with moderate effort in a given language.

An informal overview of the scope of the type system is presented next, and followed by the actual representation used in MLPE.

3.2.1 Overview

The type system is composed of

- primitive data types (integers and strings),
- structured data types (pairs and lists),
- functions,
- "IO" types, to model computations with side-effects,
- a void type, and
- (user-defined) opaque types, to model language-specific abstractions.

The primitive data types allows for representing basic unstructured data. We judged that integers and strings would suffice for basic experiments.

The structured types allow for representing structured data elements such as tuples or records by encoding those in a Lisp-like manner using pairs and lists. Pairs and lists differ in that pairs can hold elements of different types, whereas lists hold elements of a single type.

The functional types provide a language-independent means to represent and invocate functions. The IO types allow the representation of computations with possible side effects that produce values of a definite type. (In this sense it is analogous to the IO monad in Haskell.) The void type is introduced in order to account for the case that a computation may produce no meaningful value. The functional, IO and void types can be combined to model functions of an arbitrary number of arguments (of definite type), that may or may not have side effects, and that may produce a value of a definite type or may return nothing.

The opaque types allow a representation of language-dependent abstractions such as objects of an arbitrary class in an object-oriented language, or values in of an arbitrary algebraic type in a functional language. Individual opaque types are user-defined, and their interpretation is meant to be confined to the defining language, in the sense that

values of an opaque type should be accessed from other languages exclusively through the interface provided in the defining language.

3.2.2 Haskell Representation

The primary representation of values in the type system is defined using Haskell data types.

For all practical purposes, we want to be able to determine the type of any given value in the type system. This will serve as a means of documentation and will make programming in MLPE more manageable. For instance, when loading a module in the interactive environment, we would like the user to see readily the type of the values and functions defined in the module. We therefore define two Haskell types, one for representing values in the type system (`CommonType`), and one for representing types (`CommonType`). For the purpose of determining the type of a given value, we annotate the values of non-trivial types with type information.

We define the Haskell representation of types in the common type system as follows:

```
data CommonType =
     CT_Int                              -- Int
   | CT_String                           -- String
   | CT_Pair CommonType CommonType       -- (s, t)
   | CT_List CommonType                  -- [s]
   | CT_Func CommonType CommonType       -- s -> t
   | CT_IO CommonType                    -- IO t
   | CT_Void                             -- Void
   | CT_OpaqueType TypeName              -- <t>
```

The intended meaning of these types is as follows:

- Integers have the same meaning as the Haskell type `Int`.
- Strings have the same meaning as the Haskell type `String`, except that the notion of taking a string apart into characters is not reflected in the type system.

- Pairs hold two data elements of possibly different types.
- Lists hold zero or more elements of a single type.
- Functions have a definite domain and range – a function applied to a value in its domain evaluates to some value in its range. Functions of multiple arguments are assembled using function nesting (i.e. functions returning functions.)
- In the absence of IO types, functions are meant to be pure (i.e. side-effect-free) functions.
- An IO value represents a computation returning a value of a definite type, which may have side effects.
- There is only one value of type void.
- Opaque types are identified by their name (a string) in the scope of an entire project.

The type for representing values is defined as follows (the corresponding types are shown in comments):

```
data CommonObject =
    CO_Int Int                                  -- CT_Int
  | CO_String String                            -- CT_String
  | CO_Pair CommonObject CommonObject           --  CT_Pair s t
  | CO_EmptyList                                -- CT_List CT_Void
  | CO_List CommonObject CommonObject           -- CT_List t
  | CO_Func CommonType CommonType               -- CT_Func s t
            (CommonObject -> IO CommonObject)
  | CO_IO CommonType (IO CommonObject)          -- CT_IO s
  | CO_Void                                     -- CT_Void
  | CO_Obj CommonType (Ptr ())                  -- t (the opaque type t)
```

The interpretation of the constructor arguments is as follows:

- For integers and strings, the constructor argument is the represented datum.

- For pairs, the constructor arguments are the components of the pair.

- The first argument to the non-empty list constructor (`CO_List`) is the first element of the list; the second element is the tail of the list, represented itself as a list object. The empty list is represented using an additional nullary constructor (`CO_EmptyList`).

- The arguments to the constructor for functionals (`CO_Func`) are respectively the domain type, the range type and the function itself. The function argument is of IO type (`CommonObject -> IO CommonObject`) in provision for the case that a function may be defined in an imperative language, so that its evaluation needs to be performed in the IO monad.

- The arguments to the constructor for computations (`CO_IO`) are respectively the type of values produced by the computation and the computation itself.

- The arguments of the opaque object constructor (`CO_Obj`) are respectively the opaque type and the value itself. The value is represented as a "void" pointer. MLPE does not itself make any interpretation of this pointer.

3.3 Module System

All modules loaded into the system are cast into a uniform representation, which is defined using a Haskell type. This representation reflects the properties defined explicitly in the module code. The purpose of this representation is to expose the features of a module so that they can be made accessible to the other modules and made visible to the user.

The representation is instantiated by the loader when loading a module, and afterward held in the shell so that it can be queried interactively.

The properties of a loaded module are:

- the name of the module,

- the names of the modules it imports,

- the names of the opaque types defined in the module, and

- a map of the object exported by the module the object map (it associates names with values in the common type system).

The name of the module is present in the representation mainly for documentation purposes – the environment does not actually make use of it. We judged its inclusion in the module representation desirable because it forces the module name to appear in the module source code.

The names of the imported modules are used by the shell to actually load the imported modules at runtime and also to create the import context.

The names of the opaque types are, like the module name, present for documentation purposes. They could be used to provide some safety by ensuring that all opaque types involved in the type signatures of a module's objects are actually declared, but this property is not enforced.

The data structure we used for representing a loaded module is the following:

```
data Module =

   Module ModuleName                  -- name of this module

          [ModuleName]                -- names of imported modules

          [TypeName]                  -- opaque types defined in this module

          (Map_ Name CommonObject)    -- object map
```

3.4 Project Specifications

When doing development using the system, the programmer is writing modules in several languages. To hold together all those entities which pertain to the application, MLPE provides a top level entity called the *project.*

Logically, the constituents of a project are its modules. The modules are dynamically loaded into MLPE, so we consider them the *dynamic* constituents of a project. In addition to its dynamic constituents, a project has *static* constituents: the source files of the individual modules, as well as the intermediate files produced by the compilers that are needed in order to load the modules. The specification of a project is composed of the specifications of its static and dynamic constituents.

It is intended that a specification be written by the programmer to describe the project. In this sense the specification it is analogous to a "makefile".

The specification itself is written in Haskell, and dynamically loaded by MLPE. For MLPE to locate and load the project specification, we use the convention that the specification must be written in a module named `ProjectSpec`, and exported under the name `spec`. This module's source file must reside in the `spec/` subdirectory of the project home directory (see the subsection on directory structure below.)

The type we use is the following:

```
data ProjectSpec =

   ProjectSpec

       ProjectName             -- project name

       PathName                -- home directory

       ModuleName              -- target module

       [SourceFileSpec]        -- constituent source files

       [ResourceSpec]          -- resources

       [ModuleSpec]            -- modules
```

The project name appears in the specification for documentation purposes. The pathname indicates the location of the project's home directory, where its source files and resources are located. The target module is the module that gets loaded when MLPE starts up (the shell performs module chasing starting with this module.) The remaining components contain the specifications of the individual source files, resources and modules that constitute the project.

A source file's specification indicates the name of the file and the source language. The source language is identified explicitly in provision for the case that the filename's extension may not suffice to describe the nature of the file. For instance, one may want to distinguish between source files written in standard Haskell and source files that require non-standard features of some Haskell implementation. The explicit identification of the source language is used by the make tool to determine the compiler to use for compiling the source file.

The type we use for a source file specification is the following:

```
data SourceFileSpec = SourceFileSpec FileName LanguageName
```

A resource's specification is composed of

- the name of the resource,
- a resource type identifier,
- a list of constituent files (which are source files or resources), and
- a list of imported resources.

The name of the resource is the name of the resource file. The resource type describes the nature of the resource, for instance it may be a "C object file" or a "Java class file". Like the source language, it is used by the make tool to identify the tool to use to generate the resource. The constituents of the resource are those files which get transformed into the resource. For instance, when a source file is compiled into an object file, the compilation can be seen as a transformation of the source file into the object file. The imported resources are those resources that are prerequisite to the generation of the resource. For example, if a Java class makes use of another Java class, then the client class can't be compiled until the other class is compiled. This information is used by the make tool to order the generation of the individual resources.

The type we use for specifying resources is the following:

```
data ResourceSpec =

    ResourceSpec

       ResourceName          -- name of this resource

       ResourceTypeName      -- resource type

       [Name]                -- names of constituent source files/resources

       [Name]                -- name of imported resources
```

A module's specification is composed of the name of the module and the name of the source file or resource from which it is loaded:

```
data ModuleSpec =

   ModuleSpec ModuleName        -- name of this module

              Name              -- constituent source file / resource
```

An example of a project specification is given in Appendix 4, and its interpretation is given in Section 7.2.

3.4.1 Directory Structure

A pragmatic issue related to the specification of a project is the directory structure to use to store the files of a project. The following convention is used:

- the source files of a project are stored in the project's home directory;
- the resources of a project are stored in the `obj/` subdirectory of the project's home directory;
- the project specification file (named `ProjectSpec.hs`) is stored in the `spec/` subdirectory of the home directory.

The directory structure for a sample project is shown in Section 7.3.

3.5 Language Processors

We use the term "language processors" in a broad sense to refer to those entities which operate over the constituents of a project – its source files, resources and modules. MLPE defines abstractions of language processors in the form of Haskell types; it is intended that these types be instantiated in order to implement support for particular languages.

MLPE defines three classes of languages processors: compilers, linkers and loaders. The three classes differ in the nature of constituents over which they operate. Compilers and linkers operate over the static constituents of projects: compilers transform source files into resources, and linkers transform resources into other resources. They provide interfaces for connecting language implementations into MLPE. Loaders, on the other hand, take input from static constituents and realize the dynamic representation of modules.

In addition to individual language processors, MLPE defines an abstraction of a *collection* of such processors, which we call a "programming environment". (One may find the term questionable as MLPE itself is a programming environment, however we deem it suitable as the abstraction characterizes an instantiation of MLPE populated with implementations of a set of languages.) A programming environment can be viewed as an entity which operates over an entire project, in analogy with language processors which operate over its constituents.

MLPE sets up a programming environment on start-up and uses it throughout the session in the shell. It is intended that, when implementing support for a new language in MLPE, this programming environment be extended to include the new language processors.

3.5.1 Compilers

A compiler takes a source file in a given language to a resource of a given type. The representation of a compiler consists of a source language identifier, a target resource type identifier, and a compilation function. The type for representing a compiler is the following:

```
data Compiler =

   Compiler  LanguageName              -- object language
             ResourceTypeName          -- target type
                                       -- compilation function -
             (PathName ->              -- source directory
              SourceFileSpec ->        -- source file
              ResourceSpec ->          -- object file
              IO ())                   -- (returns nothing)
```

The compilation function takes as input the project's home directory, as well as the specification of the source file and target resource. The compilation function is of an IO type because the operation of invoking a compiler is indeed of imperative nature.

The compilation function returns nothing. It might have been desirable to model the output the compiler so as to report success or failure and possibly collect the textual

output of the compiler in case of failure. MLPE manages with lesser means. As the compilers are invoked in the shell which has a textual interface, the compiler's output is sent to the standard output/error stream so that it is visible to the user. As to success or failure detection, we note that success can be asserted by verifying that the time of modification of the resource file is antecedent to that of the source file.

An example of the implementation of a compiler is shown in Section 4.1.1.

3.5.2 Linkers

The next class of tools, linkers, is arguably misnamed. We adapted the term from the case where a collection of C object files are linked into a shared library. In general, a linker transforms a collection of resources of a given type into a resource of another given type.

Linkers are modeled much like compilers: the `Linker` type specifies the type of the source and target resources, and a "linking" function. The type for representing a linker is the following:

```
data Linker   =

   Linker    ResourceTypeName            -- input type
              ResourceTypeName            -- output type
              (PathName ->                -- source directory
               [ResourceSpec] ->          -- constituent resources
               ResourceSpec ->            -- target resource
               IO ())                     -- (return nothing)
```

The linking function takes as input the project home directory, the specification of the input resources as well as the target resource.

3.5.3 Loaders

A loader instantiates the internal representation of a module, as defined in Section 3.3, by loading some concrete representation of it. The concrete representation in question is a resource or source file. The representation of a loader identifies the source language of

resource types from which it loads a module, and a "loading" function. The type for representing loaders is the following:

```
data Loader    =

   Loader     ResourceTypeName              -- input type

              (PathName ->                  -- source directory

               ModuleSpec ->                -- module to instantiate

               IO (Module,                  -- the loaded module

                   ImportContext->IO()))    -- the import context handler
```

In brief, the loading function takes as input project's home directory as well as the specification of the module to load, and returns the internal representation of the module, along with the import context handler. It is intended the client of the loader (in MLPE: the shell), after invoking the loading function, will call back the import context handler and pass a valid import context to it.

To illustrate the loading process, let us take the example of the loading of a module implemented in Java (Section 4.2). The steps taken by the client (the shell) and the loader are as follows:

1. the shell invokes the loading function to initiate the loading of a module, say *ModuleX* (this takes place in the shell's first phase of module loading, see Section 3.7.2);

2. the loading function loads the module definition from its specified resource (the class `ModuleX`) through the Java class loader; this action also has the effect of creating the runtime instantiation of the module (i.e. the representation of the instance of the class `ModuleX` in the JVM);

3. the loader analyzes the module definition and creates MLPE's internal representation of the module; it returns this representation of the module along with the import context handler, which is programmed to take the action stated in step 5;

4. the shell invokes the import context handler, thereby passing the actual import context (this takes place in the shell's second phase of module loading);

5. the import context handler creates a representation of the import context in the target language (Java) and sends it to the runtime instantiation of the module.

The procedure is illustrated in Figure 6 (the numbers in the figure correspond to the steps described above.)

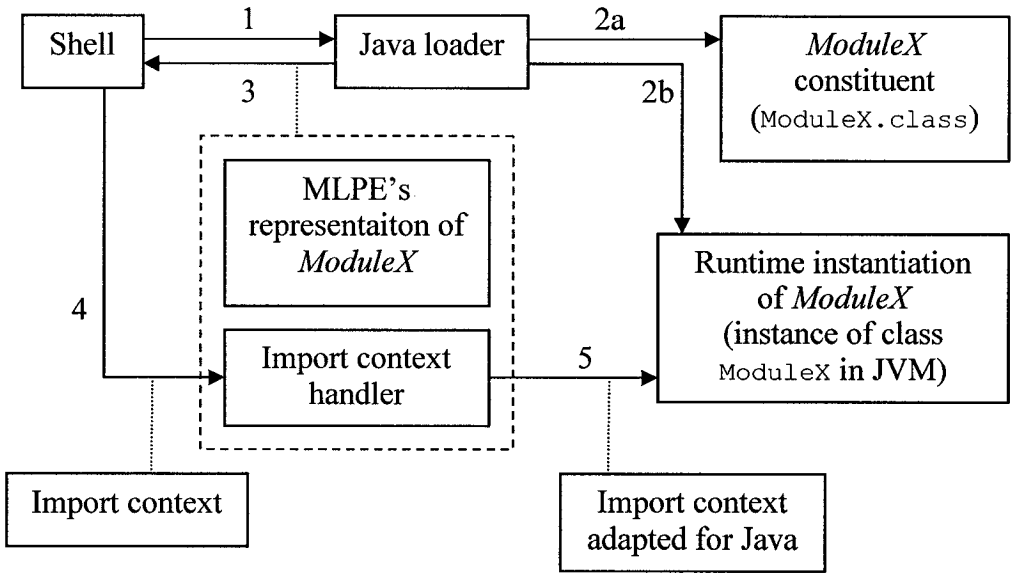

Figure 6. Loading of a Java module.

An example of the implementation of a loader is shown in Appendix 3.

3.5.4 Programming Environments

A programming environment represents a particular combination of language processors – compilers, linkers and loaders – that together can be used to process an entire project. The representation of a programming environment is composed of three maps, which indicate the tool to use for a particular combination of input and output source language or resource type. More precisely, this map indicates:

- a compiler to use for a given combination of source language and target resource type,
- a linker to use for a given combination of source and target resource types, and
- a loader to use for a given resource type.

A programming environment is represented as a value of the following type:

```
data PE = PE (Map_ (LanguageName, ResourceTypeName)      Compiler)
             (Map_ (ResourceTypeName, ResourceTypeName) Linker)
             (Map_ ResourceTypeName Loader)
```

3.6 Make Tool

The make tool invokes the compilers and linkers in order to generate the resources of a project. This operation is perquisite to loading the modules. It is performed in two steps:

- the make tool first identifies those resources which need to be generated as well as a suitable order for their generation, according to their inter-dependencies;
- it then invokes the compilers and linkers to generate the individual resources.

Programmatically, it is implemented using two high-level functions: one which builds the entire project, and one which builds a single resource. The one which builds the project is given the specification of the project and a programming environment:

```
update_project :: Project -> PE -> IO ()
```

The function first identifies those resources which need to be rebuilt because their direct constituents or imported resources were modified (this is determined by the modification time of the files.) It then identifies all the resources which may be affected, possibly indirectly, and need to be rebuilt as well. In this second step, it performs a depth first search in the entire set of resources; the output of this search is a topological ordering of the resources [CLRS01]. To be more precise, this search is done over a graph which reflects which resource affects which other resources: a resource (or source file) A affects a resource B if A imports B or B is a constituent of A.

Finally, the function initiates the generation of the individual resources in the computed order; it does by passing the specification of the resource to the other function:

```
update_resource :: Project -> ResourceSpec  -> PE -> IO ()
```

This function first determines whether the specified resource should be generated by a compiler or a linker. To determine this, it checks whether its constituents are a source file or a collection of resources. If the constituent is a source file, it looks up the

appropriate compiler in the programming environment and invokes it; if it is a collection of resources, it looks up the appropriate linker and invokes it.

3.7 Shell

In order be useful to users, MLPE needs to provide a convenient interface to expose its functionality. This is the primary intent of the shell.

In addition to providing a user interface, the shell initiates and controls the loading of modules. It also provides a mechanism to rebuild and reload the project, so that the user can see the immediate effect of any modifications made to the source files.

This section begins with a presentation of the user interface provided by the shell. A description of the mechanisms that handle the tasks of loading the modules and rebuilding the project is presented next. Finally, an overview of the programmatic organization of the shell is given.

3.7.1 User Interface

The shell exposes an interface similar to those found in conventional interpreters. It provides an interface performing the following operations:

- initiate a rebuild and reload the modules,
- list the contents of a loaded module,
- lookup values in modules and apply some functions to sample data, and
- define some bindings, so as to be able to collect the output of functions for future use.

The user controls the shell by typing commands at the prompt. The set of valid commands forms a language

The elements of the shell language are qualified names, a literal representation of basic data (integers, strings, pairs and lists,) function applications and bindings (explicit assignments). A qualified name of the form `M.o` refers to an object `o` in module `M`; a single-component qualified name (e.g. `b`) refers to a previously defined binding. Literal values and bindings refer to constants in the common type system. Applications are

written as juxtapositions as is customary in functional programming (e.g. `f x` applies function `f` to argument x).

The shell language is implemented using the prototyping tool (Chapter 5). For reference, the definition of the language is included in Appendix 1.

In addition to the shell language defined above, the shell recognizes a number of high level directives. These directives all begin with a colon sign (the same convention used in Hugs and GHCi.) The recognized directives are the following:

:reload	rebuild the project and reload the modules
:quit	quit the shell
:modules	prints a listing of the names of the loaded modules
:bindings	shows the set of bindings

For convenience, all commands can be abbreviated to their first letter (e.g. `:q` is equivalent to `:quit`.)

Sample sessions in the shell are shown in Chapter 7.

3.7.2 Module Loading in the Shell

The loader abstraction provides an interface for loading a single module. Let us recall the interface defined by the loader abstraction for loading a module; loading a module is done in two steps:

1. first the client invokes the loading function and thus obtains a representation of the module, along with the import context handler;
2. the client then provides the module with its import context by passing it to the import context handler.

Of course, in the programming environment, a *set* of modules must be loaded. Moreover, for the purpose of debugging individual modules, it is desirable that only a subset of the modules of a project be loaded at once. There are two difficulties about loading a collection of modules in the environment:

1. one is to identify those modules which must be loaded;

2. the other is to "link" the modules together, by providing each module with its import context.

The implementation is based on the technique of *module chasing*: one module is identified as the target module; once it is loaded, all of its imported modules are loaded in turn, and, recursively, their imported modules are loaded as well. The target module is identified in the project specification.

The shell performs loading in two phases:

phase 1. it first loads all the modules by performing module chasing: it first loads the target module by invoking the appropriate loader; it then inspects the list of imported modules from the module definition, and recursively invokes the module chasing mechanism to all the imported modules;

phase 2. once the modules are loaded, it assembles the import contexts of the individual modules, and feeds them back to the individual modules by passing them to the respective import context handlers.

3.7.3 Rebuilding a Project

The shell supports the high-level operation of initiating a rebuild of the project and reloading the modules. The task of building the project is handled in the make tool; that of reloading the modules is handled in the shell.

It is difficult in general to unload and reload modules dynamically. We had experimented with a scheme for supporting this, but it proved to be very difficult to implement correctly for certain languages. For instance, in Java, reloading classes is an involved topic in itself (see e.g. [LB98]).

Fortunately, an easy alternative exists. It is simply to restart the MLPE process; on start-up, MLPE builds the project anew and loads the modules. This is implemented by using a special exit code to signal that the process should be restarted – this exit code is handled by an external script, which starts the process every time this exit code is returned.

3.7.4 Organization of the Shell

The central data structure of the shell is a representation of its entire state. It holds the specification of the current project, the programming environment (i.e. the set of tools), the set of loaded modules (that we call the *run-time context*), and the current bindings.

On start-up, the shell is programmed to initiate a build of the project. Afterward, it repeatedly reads a command from the input stream, performs the requested operation, and commits any change to the state.

3.8 Startup

On startup, MLPE loads the specification of the project and starts the shell.

As the project specification is written in Haskell, it must be compiled before it can be loaded. MLPE invokes GHC, the Haskell compiler, over the specification file, and then uses the Haskell dynamic loader to load the specifications.

Chapter 4 Black-Box Language Implementations

This chapter reports the implementation of the support for three general purpose languages in MLPE using available implementations.

The properties of the object language and extrinsic properties of the implementation indeed affect the implementation, but the inner workings of the particular implementation do not. In this sense, these implementations are used as black-box language processors.

The primary aim in implementing support for an object language is to provide it with an access to the common abstractions of MLPE (its module and type system) so that modules can be implemented in the object language and can thus take part in mixed-language applications. It is achieved by writing a library in the object language for interfacing MLPE. In particular, the language is not extended with new syntax. The nature of the support for MLPE provided in each language can be seen from the source code of the sample modules in Appendix 5 through 7.

Implementing support for a language in MLPE is a moderately extensive programming exercise. It involves programming an interface to MLPE in the object language, and extending MLPE to access this interface. In the object language, one must

- implement a representation of values and types in the common type system;
- define a format for describing a module (in terms of the modules it imports, the symbols it defines, etc.);
- define an entry point for MLPE to fetch the module description;
- define a representation of the import context, and define an entry point for MLPE to pass the import context to the module.

As to the extension to MLPE itself, it consists of

- implementing wrappers for the compilers and linkers if such tools are to be involved;
- implementing a means to access the object language's representation of the common type system;

- implementing a loader, which involves fetching the module description and adapting the import context to the object language.

These steps are carried out for three languages: C, Java and Haskell. We estimate that these three languages form a sample that is sufficient to uncover the difficulties of the task.

4.1 C

Implementing support for modules in C is facilitated by the fact that Haskell provides an extensive interface for accessing C (Haskell's Foreign Function Interface, or FFI [FFI]).

4.1.1 C Compiler and Linker

GCC is used to generate object code, and is used again to link the object file into a shared library that can be dynamically loaded in the environment.

It is straightforward to connect the actual compiler or linker into its abstract representation: we just invoke it using a shell command with appropriate options. For compilation, the options to specify are the name of the source and object file. It is also in this scope that the resources are sent to `obj/` subdirectory of the project home directory, where all the resources are stored (see Section 3.4.1). The wrapper for the C compiler is implemented as follows:

```
gcc =

   Compiler "C"                    -- source language
            "C object file"        -- target resource type
                                   -- compilation function:
            (\ dir ->                                  -- home directory
             \ (SourceFileSpec src_name lang) ->       -- source file
             \ (ResourceSpec obj_name _ _ _) ->        -- target resource
                do let cmd = "gcc -c " ++ (dir ++ src_name)
                             ++ " -o " ++ (dir ++ "obj/" ++ obj_name)
                   putStrLn cmd
                   system cmd
                   return () )
```

The wrapper for the linker is, but it invokes GCC with different options. In the command line, it specifies the object files to link and the target dynamic library. (Again, all these resources are sent to the `obj/` directory.) It also includes an option to specify that the output is a shared library.

4.1.2 Representation of the Common Type System in C

There are two parts to the representations of types and values in the common type system in C:

- the actual representation using C's data structures, and
- a procedural interface to access this representations.

This separation provides data abstraction: the user needs only know about the procedural interface; also, the part in Haskell accesses the C representation through this interface only.

4.1.2.1 Internal Representation

The internal representation of values and types in the types system is defined using two C `struct`'s: one for values and one for types. In order to account for the different

constructors (such as `CT_Int`, `CT_String`, etc.), a designated structure field, the "type tag", is included in the data structure – to each constructor is associated a value for this field.

The individual arguments to the constructors are stored in additional fields. These fields are typed as void pointers, and we will assign an interpretation of these pointers for each constructor. The two structures are defined as follows:

```
struct common_type {
  common_type_tag type;
  void *arg1, *arg2;
};
struct common_object {
  common_type_tag type;
  void *arg1, *arg2, *arg3;
};
```

The first, second, and third arguments of any constructor are stored in the fields `arg1`, `arg2`, `arg3` of the corresponding structure (up to the arity of the constructor in question). For instance, for a pair object (a value in the constructor `CO_Pair`), the representations of the first and second components of the pair are stored in the `arg1` and `arg2` fields, respectively, of a structure of type `struct common_object`.

For convenience, a type synonym is defined:

```
typedef struct common_object CommonObject;
typedef struct common_type CommonType;
```

In all places of the code, this synonym is used instead, and let functions manipulate objects of type written as `CommonType` and `CommonObject`.

The correspondence between the type of an argument to a Haskell constructor and the type of the corresponding structure field is as follows:

Haskell Type	C type
`CommonType`	`CommonType*`
`CommonObject`	`CommonObject*`
`Int`	`int`
`String`	`char*`
`Ptr()`	`void*`
`TypeName`	`char*`

Table 1. Correspondence between Haskell types and C types

This table does not account for the functional arguments to the constructors `CO_IO` and `CO_Func`; the type of these arguments are, respectively, `IO CommonObject` and `CommonObject -> IO CommonObject`. Our representation of these functional arguments must allow programmers to define functions in C, and to use functions defined in other modules.

The case of `CO_IO` has a simple solution. An object of this type is simply a procedure taking no argument and returning a value of some type. The functional argument, of type `IO CommonObject`, can therefore be modeled with a function having this signature:

```
CommonObject* f();
```

The case of `CO_Func` is more intricate. The first problem concerns the number of arguments to the function. It is definitely desirable to be able to define functions using C's signature scheme: one should be able to represent a *n*-argument function object using a *n*-argument C function. In C, however, there is no notion of partial application of a function, as is common in functional programming (i.e. the feature that a *n*-argument function, when supplied the first argument, evaluates to a function of *n*-1 arguments; note that the common type system has this feature). We will not attempt to model this feature in C.

The correspondence we adopt is the following. For representing the functional argument to a function object of *n* arguments, the function with the following signature is used:

```
CommonObject *f(CommonObject *a1, CommonObject *a2
                ... CommonObject *an)
```

A further complication is that in C, the distinction between pure functions and functions having side effects does not exist either. Functional arguments to the constructor are insensitive to this. For instance, for a function of type `Int -> IO Int` and a function of type `Int -> Int`, the functional argument of either constructor would be of type:

```
CommonObject *f(CommonObject *a1, CommonObject *a2);
```

and in either case, the function is expected to return the C representation of an object of type `Int`, not `IO Int`.

4.1.2.2 Procedural Interface

There are two parts to the procedural interface: constructors, for instantiating the C representation, and inspectors for querying the C representation.

Having defined the correspondence between Haskell types and C types, the signatures of the constructor functions for the C representation of types and objects follow directly:

```
// types
CommonType* ctInt();
CommonType* ctString();
CommonType* ctPair(CommonType *a, CommonType *b);
CommonType* ctList(CommonType *a);
CommonType* ctFunc(CommonType *a, CommonType *b);
CommonType* ctIO(CommonType *t);
CommonType* ctVoid();
CommonType* ctType(char *type);
// objects
CommonObject* coInt(int i);
CommonObject* coString(char *s);
CommonObject* coPair(CommonObject *h, CommonObject *t);
CommonObject* coList(CommonObject *h, CommonObject *t);
CommonObject* coEmptyList();
CommonObject* coFunc(CommonType *t1, CommonType *t2, void* f);
CommonObject* coIO(CommonType *t, CommonObject*(*f)(void));
CommonObject* coVoid();
CommonObject* coObj(CommonType *t, void *ptr);
```

Note that, in the constructor `coFunc()`, the functional argument must be left as `void*` because the signature of the supplied function argument depends on the arity of the function.

The inspector functions give access to the individual constructor arguments given the C representation of an object or type. Before one can access these arguments, however, it is essential to be able to identify the constructor of that particular object or type. The following functions expose the type tag of types and values:

```
common_type_tag ctGetTypeTag(CommonType *o);

common_type_tag coGetTypeTag(CommonObject *o);
```

Their return value is one of the following:

```
common_type_tag CT_Void      =0;

common_type_tag CT_IO        =1;

common_type_tag CT_Func      =2;

common_type_tag CT_Int       =3;

common_type_tag CT_String    =4;

common_type_tag CT_EmptyList =5;

common_type_tag CT_List      =6;

common_type_tag CT_Pair      =7;

common_type_tag CT_Obj       =8;
```

The same set of values is used for identifying constructors of both values and types. For instance, `CT_Int` is the type tag the two constructors `CT_Int` and `CO_Int`. This is true about all these values, except `CT_EmptyList`, which identifies the constructor for the empty list object but does not correspond to any type constructor.

When the constructor is identified, the client can retrieve the constructor arguments in a type-safe manner. For type constructors, most constructors have one or two type arguments, and we provide two generic functions to retrieve them:

```
CommonType* ctGetTypeArg1(CommonType *o);

CommonType* ctGetTypeArg2(CommonType *o);
```

The only type constructor having an argument of a different type is that for an opaque type, which has a string argument. The following function allows client to retrieve the name of the opaque type:

```
char* ctGetTypeName(CommonType *o);
```

The arguments to value constructors are more varied in type. The following excerpt for the source code lists the signatures of the inspector functions and give, in comments, the type of objects to which the individual functions apply.

```
int                 coGetInt(CommonObject*);        // CO_Int
char*               coGetString(CommonObject*);     // CO_String
CommonObject*       coGetLeft(CommonObject*);       // CO_Pair
CommonObject*       coGetRight(CommonObject*);      // CO_Pair
bool                coIsEmpty(CommonObject*);       // CO_List
CommonObject*       coGetHead(CommonObject*);       // CO_List
CommonObject*       coGetTail(CommonObject*);       // CO_List
CommonType*         coGetTypeArg1(CommonObject*); // CO_Func, CO_IO, CO_Obj
CommonType*         coGetTypeArg2(CommonObject*); // CO_Func
void*               coGetFunc(CommonObject*);       // CO_Func
IO_Func             coGetIO(CommonObject*);         // CO_IO
void*               coGetObj(CommonObject*);        // CO_Obj
```

where the type `IO_Func` is defined as a type synonym:

```
typedef CommonObject* (*IO_Func)(void);
```

4.1.3 Accessing the C Representation from Haskell

Once the representation of types and values is defined in C, we can make our way to access it from Haskell, so as to be able to exchange values and types with a module written in C with the programming environment. There is a canonical way to access C from Haskell: through Haskell's Foreign Function Interface (FFI).

Programmatically, we wish to define marshaling functions over types and values. In Haskell, C objects representing types and values are considered a foreign representation of the equivalent Haskell types, so the type of objects use to refer to the foreign objects are `(Ptr CommonType)` and `(Ptr CommonObject)`. (This is a convention of the FFI, that a foreign representation of a type *t* should have type `Ptr` *t*.) For exporting values to C, we define the following two functions:

```
marshalType          :: CommonType   -> IO (Ptr CommonType)

marshalObject        :: CommonObject -> IO (Ptr CommonObject)
```

(We defined these as IO functions because they need to invoke the constructor functions in C, which is an imperative operation.) For importing values from C, we define the following two.

```
unmarshalObject      :: Ptr CommonObject -> IO CommonObject

unmarshalType        :: Ptr CommonType   -> IO CommonType
```

Marshaling types is straightforward. For exporting types to C, the marshaling function simply invokes the appropriate C constructor and returns the pointer. For importing types from C, the function first looks up the type tag and, in accordance with the identified type constructor, fetches the constructor arguments using the inspector functions defined in C, and returns a `CommonType` value.

Marshaling values is more involved than marshaling types. Handling primitive and structured data types bears no particular difficulty – it is similar to marshaling types. Again, the difficulty arises with handling functions.

4.1.3.1 Exporting Functions to C

The difficulty with exporting functions to C is to design a generic mechanism to export a *n*-argument function.

Our strategy is to define in Haskell a number of types that correspond to the type of the functional argument to `CO_Func` for functions with different number of parameters.

In short, the steps performed by the marshaling function are the following:

1. identify the number of parameters to the function,
2. instantiate a Haskell function with the appropriate number of parameters which takes its parameters and returns a value using the C representation,
3. obtain a C wrapper function around the Haskell function, and finally
4. invoke the C constructor (`CO_Func`,) passing the C wrapper function for the functional argument.

The individual steps are straightforward functional programming.

The Haskell types used to represent the functional argument to `CO_Func` with functions of different arguments are defined as follows (for functions of one or two parameters):

```
type Func1 = Ptr CommonObject
          -> IO (Ptr CommonObject)
type Func2 = Ptr CommonObject -> Ptr CommonObject
          -> IO (Ptr CommonObject)
```

We use the FFI to obtain a C function that is a wrapper around the Haskell function. It is programmed by declaring "wrapper-generating" functions:

```
foreign import ccall "wrapper" mkFunc1 :: Func1 -> IO (FunPtr Func1)
foreign import ccall "wrapper" mkFunc2 :: Func2 -> IO (FunPtr Func2)
```

Applying these functions to a Haskell function returns a pointer to the wrapper function.

4.1.3.2 Importing Functions from C

For importing functions defined in C, the marshaling function is also problematic. The first issue is about partial function application – what does a n-argument function return when supplied fewer than n arguments? We did not model the partial application of functions in C; however, we clearly need this feature when taking the function into the environment. In order to do it we need to "defer" the passing of arguments to the actual C function until *all* the arguments are supplied. This can be done by accumulating the arguments in a list and passing them when doing the inner call. The following function applies this technique:

```
defer :: CommonType          -- partial type
      -> Ptr CommonObject    -- C representation of the function object
      -> CommonObject        -- accumulated argument list
      -> CommonObject        -- (partial-application-able) function
defer (CT_Func s t) p parms
   = CO_Func s t (\p0  -> do return (defer t p (CO_List p0 parms)))
defer (CT_IO t) p parms
   = CO_IO t (do parms_c <- marshalObject parms
                 v_c     <- applyProc p parms_c
                 v       <- unmarshalObject v_c
                 return v)
defer t p parms
   = CO_IO t (do parms_c <- marshalObject parms
                 v_c     <- applyProc p parms_c
                 v       <- unmarshalObject v_c
                 return v)
```

Another issue is to design a generic mechanism to call a n-argument C function from Haskell. This time, we will write most of the solution in C.

The key is to write a C function for applying a function to an arbitrary number of arguments – the arguments are passed in a list rather than separate arguments. The function in question has the following prototype:

```
CommonObject *applyProc(CommonObject *f, CommonObject *args);
```

The first parameter is the function object, the second parameter is the list of arguments (encoded in a `CO_List` object), and the return value is the value returned when doing the actual call. The function counts the number of arguments in the list, and does the actual call through one of the many specialized functions which handle the call for a given number of parameters. For instance, the function for handling calls to functions of one argument is defined as follows:

```
CommonObject* applyProc1(CommonObject *f, CommonObject *arg0) {
  CommonObject* (*p) (CommonObject*) =
       (CommonObject* (*) (CommonObject*)) (coGetFunc(f));
  return p(arg0);
}
```

4.1.4 Module Entry Point

The entry point to a C module is defined as a function of the following type:

```
ModuleSpec describeModule();
```

where the C type `ModuleSpec` is just a synonym for the C type `CommonObject*`. It is expected that the `describeModule()` function returns a description of the module in the format defined in the next section. When loading a module, the loader looks up the `describeModule` symbol from the dynamic library into which the module is compiled, and invokes the function to obtain the description of the module.

4.1.5 Module Description

There are two aspects to the description of a module. One is the format in which the module description is encoded. The other is the interface provided to the user to compose the module description. We discuss these two aspects in turn.

4.1.5.1 Encoding of the Module Description

We will save much programming by using the data structures we already have – the common types system – to encode a module definition. There are four components to a definition, which are represented individually as follows:

the module name is represented in a value of type `CT_String`,

the names of the imported modules is represented as a list of strings, or a value of type `CT_List CT_String`,

the names of the opaque types defined in the module, also represented as a list of strings, and

the symbols defined in this module represented as a list of pairs, the first component of a pair being the symbol encoded as a `CO_String` value, and the second component being the value itself.

The entire module definition is held together using pairs; in Haskell syntax, the format of a module definition is represented as follows:

```
(CO_Pair (CO_Pair (CO_Pair module_name
                           import_names)
                  opaque_type_names)
         values)
```

4.1.5.2 Procedural Interface for Describing a Module

The final element is to define a set of C functions to facilitate the definition of a module, so that the user does not have to bother with the actual encoding used to pass this information to the system. It is intended that these functions be called from within the `describeModule()` function. The top-level function for defining a module is the following:

```
CommonObject *moduleSpec(char *module_name, CommonObject *imports,
                         CommonObject *types, ...);
```

The user specifies the name of the module as a C string; for the names of the imported modules and opaque types, the user is expected to use following two functions:

```
CommonObject *imports(char *module0, ...);
CommonObject *types(char *t0, ...);
```

where the names of the modules or types are given separately as C strings, and the last argument is a NULL pointer. (Unfortunately, we need some explicit parameter to signal the end of that list, because C provides no way to detect the end of the parameter list otherwise.) The remaining arguments to the `moduleSpec()` function list the objects defined in the module. Again, the last argument to `moduleSpec()` must be a NULL pointer. For specifying the individual (symbol, object) pairs, the system provides this last function.

```
CommonObject *entry(char *name, CommonObject *obj);
```

To illustrate the use of these many functions, suppose a module `A` imports the modules `B` and C, defines two string objects `s` and `t`, and no opaque type. The module code would include a function like the following:

```
ModuleSpec describeModule() {
   return moduleSpec(
              "A",                          /* module name */
              imports("B", "C", NULL),      /* imported modules */
              types(NULL),                  /* opaque types */
                                            /* exported symbols- */
              entry("s", coString("Rien ne sert de courir.")),
              entry("t", coString("Il faut partir à point.")),
              NULL);
}
```

4.1.6 Passing the Import Context to a C Module

Let us recall that the import context is a function that, given the name of a module and symbol defined in that module, returns the associated value:

```
type ImportContext = ModuleName -> Name -> IO CommonObject
```

Making the import context accessible in C involves (1) taking this Haskell function to a C function, and (2) passing the function to the C module so that it can make calls to it.

In C, we would like the import context to be accessible through an interface like this one:

```
CommonObject *import_context(char* module_name, char* object_name);
```

In Haskell, we can express its type as follows:

```
type ImportContext_C = CString -> CString -> IO (Ptr CommonObject)
```

where `CString` is a type defined by the FFI for representing C-strings. To take the import context (of type `ImportContext`) to this later type (`ImportContext_C`), we just instantiate a new function that takes its arguments as C strings, converts them to Haskell

strings, applies these arguments to the import context, and marshals the returned object to C. This gives us a Haskell function, though, for which we must obtain a C wrapper using the FFI.

The remaining issue is to supply the context function to the module. Our strategy for doing this is to let the module define a variable of predefined name, and have the loader assign the function to it. For convenience, we define a type synonym to refer to the type of the import context in C; we call this type synonym `RT`:

```
typedef CommonObject* (*RT)(char*, char*);
```

Then, a module is expected to define a variable with the predefined name `rt`, of this type `RT`: every C module should include the following declaration:

```
RT rt;
```

The loader will obtain the address of this variable from the operating system's dynamic loader and assign the import context function to this variable. In fact, it is not possible to do assignments through the FFI, so we need to do the assignment in C, which we do using the following function:

```
void setRtEntry(void (**rt_var)(void), void (*rt_func)(void)) {
  *rt_var = rt_func;
}
```

4.1.7 Organization of the Loader

We can finally take a look at the loader itself and see how it functions. The steps it performs are as follows:

using the C dynamic loader, it opens the C library which is the resource corresponding to the module to loaded;

it then looks up the `describeModule` symbol from the library, and invokes the corresponding function;

it analyzes the returned module definition to extract its components (thereby importing all the objects), and instantiates the `Module` object accordingly, which is the run-time system's internal representation of a loaded module;

it looks up the `rt` variable from the loaded library, which is meant to be assigned the import context function;

finally, it returns the `Module` object to the system (to the shell, in fact), along with the import context handler – the import context handler (which receives the import context function as an argument) is programmed to export the import context function to C and assign it to the `rt` variable.

4.2 Java

Compared to C, Java is more "remote" to the programming environment. C is the underlying implementation technology of GHC and is part of the entire runtime system and process semantics; in contrast, Java has is own distinct runtime system – the Java Virtual Machine (JVM). In this sense, implementing support for Java may be more representative of the general task of implementing support for a language using an existing language implementation.

Implementing support for Java involves interfacing the JVM and interacting with the class loader to get the modules loaded.

In Java, we define a representation of the common type system using a set of classes, so that each value or type in the type system can be represented as an object. We define a convention for defining a module using a class: to define a module, the programmer defines a new class with some predefined properties. It is also through his class that the import context is made available to the module.

We begin this section with an overview of programming with the JNI, and then proceed with the implementation of module support in Java.

4.2.1 Interfacing the JVM

Java defines a standard interface for interacting with the JVM from C: the Java Native Interface, or JNI. This interface allows a C program to instantiate the JVM, load classes, instantiate objects and call methods, etc. It also supports the definition of Java methods in C (so-called native methods.)

There is no standard way to access the JVM from Haskell, however, as is the case for C with the FFI. There does exist an Haskell-Java interface, code-named `jvm-bridge`[1]. It is a fairly large and evolved system in itself. Unfortunately, however, hardly any documentation is available. Another option would be to use Lambada [MF00], which simplifies the interfacing between the Haskell and the JNI. We choose to use the JNI directly over this latter option because it gives the most flexibility. As the JNI is a C interface, we import its functions using the FFI.

The first step in using the Java runtime system is to instantiate the JVM. The instantiation function returns a pointer to a structure of type `JNIEnv`. This structure contains a set of pointers to the many functions of the JNI. For instance, the *instanceof* predicate is defined as a member of this structure, as follows:

```
/* jni.h */
struct JNINativeInterface_ {
    ...
    jboolean (JNICALL *IsInstanceOf)
      (JNIEnv *env, jobject obj, jclass clazz);
    ...
}
```

(where `struct JNINativeInterface_` is actually a type synonym to `JNIEnv`.) As a practical matter, the FFI does not allow access to individual fields of C structures. We therefore need to write a separate C function to do the field access. To complete the example, we define a C wrapper for the *instanceof* predicate function as follows:

```
jboolean isInstanceOf(JNIEnv *env, jobject obj, jclass clazz) {
  return (*env)->IsInstanceOf(env, obj, clazz);
}
```

We can then import this function using the FFI and access the feature by passing the `JNIEnv` variable as a parameter. We write such wrapper functions for all the functions of the JNI that we use.

Unfortunately, there is one issue that we don't handle neatly using the JNI. In a few instances, we need to pass a dynamically created C function to Java. In Java, however there is no simple and clean mechanism to represent a C pointer or a dynamically instantiated C function. In the lack of an appropriate mechanism, we choose to short-circuit the safety requirement and opt for the simplest mechanism that will work: we simply represent a function pointer as a Java integer.

4.2.2 Java Compiler

We use Jikes as the Java compiler in our experiments. We chose Jikes in favor of the standard compiler of the JDK (`javac`), because it compiles much faster. This translates in better response time in the environment when doing a rebuild, which is among our stated objectives.

We note that it is straightforward to change the compiler that is used here. The two compilers (Jikes and `javac`) have a similar command-line interface, work over the same input language and produce binaries in the same format. It would be easy as well to extend the shell to switch between programming environments, and provide different programming environments with different Java compilers.

4.2.3 Representation of the Common Type System in Java

In Java, the canonical way to express data types such as `CommonObject` and `CommonType`, whose many constructors represent data elements of various nature, is to define an abstract class for each of the types and derive a concrete class for each constructor. This is what we do here.

Representing types from the common type system is straightforward. We define an abstract class `CommonType` with no method. Each concrete class is derived from `CommonType` and provides a public constructor with arguments analogous to those of the corresponding Haskell constructor; the arguments are stored in private class fields, and made visible through public accessor methods. The signatures of their constructors are as follows:

```
public CT_Int();

public CT_String();

public CT_Pair(CommonType x, CommonType y);

public CT_List(CommonType x);

public CT_Func(CommonType x, CommonType y);

public CT_IO(CommonType x);

public CT_Void();

public CT_OpaqueType(String s);
```

We proceed similarly with values. We define an abstract base class `CommonObject` with no method. For primitive data types, we use the corresponding Java types to hold the data. Their constructors are as follows.

```
public CO_Int(int x);

public CO_String(String x);
```

The constructor for pairs is as follows:

```
public CO_Pair(CommonObject l, CommonObject r);
```

We use two classes to represent lists, which correspond to the two constructors in the Haskell representation:

```
public CO_List(CommonObject h, CommonObject t);

public CO_EmptyList();
```

To reflect the fact that the empty list is a valid list, we make `CO_EmptyList` a subclass of the class `CO_List`.

As usual, functions are the problematic case. It is clear that a function will be defined using a method, and that this method should also have a predefined signature. There is an option of using inheritance to enforce the use of a proper signature. There is also the question whether the method should be defined along the `CO_Func` object itself, or in a separate object passed to the constructor. After experimenting with various schemes, we opted for the most concise solution. Our solution is to have a constructor taking the two type arguments:

```
public CO_Func(CommonType x, CommonType y)
```

and let the client provide the function by adding an inline method when instantiating the object. The added method must be declared public and non-static; it must have name `func`, take arguments of type `CommonObject` and return a value of type `CommonObject`. For instance, the identity function over integers is implemented as follows:

```
new CO_Func(new CT_Int(),
            new CT_Int()) {
   public CommonObject func(CommonObject x) {
      return x;
   }
};
```

As in C, the representation of pure functions and functions with side effects make use of the same method signature. For instance, the method in the above definition could as well have been used to define a value of type `Int -> IO Int` rather than `Int -> Int` – the only difference would be in the type argument.

For representing procedures of no arguments (i.e. values of type `IO t`), we write an additional class, whose constructor is the following.

```
public CO_IO(CommonType _type)
```

The functional component of this object is meant to be supplied in the same way as for functions, using the same convention for the method signature. For example:

```
new CO_IO(new CT_Int()){
   public CommonObject func() {
      return new CO_Int(x);
   }
};
```

We define a class with nullary constructor for the void object:

```
public CO_Void()
```

The representation of opaque types is problematic. In MLPE, the datum of an object of an opaque type is represented by a pointer. In Java, we would like to allow any object to be represented as a value of an opaque type. In order to have a uniform representation, we store a pointer (encoded as a Java integer) to the datum – in the case of an opaque object defined outside Java, this pointer is just the pointer provided by the runtime system; in the case of a Java object, it is the pointer to the JVM's representation of the object. For the runtime system to instantiate an opaque object with just the pointer, we provide this constructor which takes the pointer as parameter:

```
public CO_Obj(CT_Type t, int ptr);
```

For a Java module to define opaque objects, we provide this alternate constructor:

```
public CO_Obj(CT_Type t, Object o);
```

The latter constructor makes use of a native method to obtain a pointer to the object. The pointer itself is stored in a private field of integer type. The pointer itself is not meant to be manipulated in Java, so we provide no accessor to it. We provide a function, however, to access the object, in case the datum is a Java object:

```
public native Object getObject();
```

It is a native method that casts the integer representation of the pointer to a pointer to a JNI object.

4.2.4 Accessing the Java Representation from Haskell

The process is very similar to what we have done in C. Again, we write the marshaling and unmarshaling functions over types and values. The main difference is that, instead of calling the constructors or accessor functions directly, we need to access the methods through the JNI. Therefore, we need to pass the JNI environment pointer to every function. The function signatures are as follows.

```
marshalType     :: JNIEnv -> CommonType    -> IO JCommonType
marshalObject   :: JNIEnv -> CommonObject  -> IO JCommonObject
unmarshalType   :: JNIEnv -> JCommonType   -> IO CommonType
unmarshalObject :: JNIEnv -> JCommonObject -> IO CommonObject
```

The handling of types bears little difficulty. For marshaling a type to Java, the function merely invokes the appropriate constructor. However, calling a constructor through the JNI primitives involves a number of steps:

obtain a handle on the class from the JNI,

obtain a handle on the constructor from the JNI – in JNI, handles to constructors are obtained like ordinary methods, and these handles are called method IDs

marshal any constructor arguments to Java,

wrap these arguments into "jvalues", which are type used to represent arguments of any type supplied to method through the JNI,

bring the "jvalues" together in an array, and finally

invoke the constructor.

To illustrate, the code that marshals the type `(IO t)` is as follows:

```
(CT_IO t) ->     do t_j <- marshalCommonType env t
                    cls <- findClass env "rt/CT_IO"
                    mid <- getMethodID env cls "<init>"
                                               "(Lrt/CommonType;)V"
                    t_jv <- makeObjectJValue t_j
                    obj <- withArray [t_j] $ \ args ->
                              newObjectA env cls mid (castPtr args)
                    return obj
```

For importing types from Java, the first issue is to identity which constructor is involved. We use the class name of the type object to do the dispatch. Afterward, the function

invokes the accessor function to get the constructor arguments, applies the unmarshaling function as needed over the arguments, and returns the value.

Marshaling values to Java is straightforward for most of the constructors and is generally similar to marshaling types; the difficult part is, as usual, to handle functions and IOs.

4.2.4.1 Exporting Functions to Java

Exporting a function to Java raises three issues: we must find a way (1) to actually pass the function to Java, (2) to handle the varying number of arguments, and (3) to realize a function object in Java to wrap the function. In brief, our solution is to export the function to C, and use a native method to call it.

The first part of our solution is to prepare a Haskell function that takes arguments and returns values in the types of Java; in order to defer the multiple-argument issue, we arrange this function to take a *list* of arguments, so as to have a fixed signature regardless of the actual number of arguments. This is the function that we are going to export to C; it will be convenient to have a name to refer to its type, so let us define the following:

```
type ExportedFunction_Java =
            JCommonObject       -- a list (CO_List) of arguments
      -> IO JCommonObject       -- return value
```

where `JCommonObject` is a type we defined for pointers to the JVM's representation of objects of the class `CommonObject`. The function of this type is merely a wrapper around the original function that also does the marshaling over the argument and return value and analyzes the argument list to call the inner function with its the individual arguments.

We first take this function to C using the FFI. The next task is to design a class to wrap instances of such functions. The class should have the same interface as ordinary function objects, so we define it as a subclass of the class for defining functions in Java, `CO_Func`. We call the derived class `CO_ImportedFunc`. We define a constructor identical to the one of its base class, which takes the two type arguments:

```
public CO_ImportedFunc(CommonType s, CommonType t)
```

Objects of this class will have to call the function, so it would be desirable to store the function in an instance field. Again, the function pointer is passed as an integer.

```
public void setFuncPtr(int i) {
    fun_ptr = i;
}
```

The remaining problem is to provide an interface for doing the call. We already know the function signatures; depending on the number of arguments, the signature to use is one of the following:

```
public CommonObject func();
public CommonObject func(CommonObject a1);
public CommonObject func(CommonObject a1, CommonObject a2);
...
```

Since these methods have to do the actual call to the C wrapper function, they must be native methods. The methods obtain the integer field from the object, cast it to the proper function type, and calls the method with the arguments disposed in a list.

4.2.4.2 Importing Functions from Java

Importing functions from Java is more straightforward. It is fairly similar to what we have done in C. The central difficulty, again, is to recover the partial application feature. But we already know the solution, and merely apply it again – that is, we adapt the `defer` function to work with Java. The main difference lies in how the method actually gets called – and this is straightforward JNI programming. The facts that the method is defined as an inline method, and that the number of argument varies with the function type, are not problematic. We can obtain the actual class of the object from the JNI (this class is actually different from function to function), and then lookup the `func` method by requesting a method with signature that matches the function type. The marshaling function must, as usual, marshal the function arguments, put them in an array and supply it to the method when doing the call, and unmarshal the value returned by the method.

4.2.5 Module Description and Entry Point

We need to identify some static program entity that will serve as an entry point to modules written in Java. We can bind the module description to a class field or let a static method return it, but at any rate, the field or method has to lie in the scope of a class. It appears more logical, then, to think of that class as the provider of the module interface. This is what we do: for a module of a given name, the system will try to load class of the same name. The class should implement a method with a predefined name which returns a description of the module in a predefined format. As we have done in C, we choose to encode the module descriptor inside a value in the common type system, and using the same type as in we did in C, so as to realize an economy on design and low-level programming. The method should have the following signature:

```
public CommonObject describeModule();
```

We provide a method to help construct the module descriptor from its components. In order to have this method in scope when defining a module, we define this method in a class, called `Module`, meant to serve as the base class for all module interface classes. The method we supply has the following signature

```
protected CommonObject spec(String module_name,
                            String imports[],
                            String types[],
                            CommonObject  values[])
```

The first three arguments are strings or arrays of string, but the fourth is an array of values, each value representing an entry in the object map, and being encoded as a (key, value) pair in the common type system. In order to make it more concise to express these entries, we define a method in the base class to define them:

```
protected static CommonObject entry(String s, CommonObject o)
```

4.2.6 Exporting the Import Context to Java

To represent the import context in Java, we define a class called `RT`. The class holds a pointer to the C function that wraps the import context in an instance field, and provides a native method (the method `sym`) to access the C function. The `Module` object holds an

`RT` object in an instance field named `rt`. A module can then access a foreign symbol, say symbol *s* in module *m*, by invoking `rt.sym(`*m*`,`*s*`)`.

As a matter of design, there is a reason why we wrap the import context in a separate object rather than adding this feature to the `Module` class directly. Logically, we want to be able to implement a module in Java using a *set* of classes. If the import context was provided as a method of the `Module` object, we would need to pass the `Module` object around, which doesn't match the intended meaning of the `Module` object as a module descriptor. In contrast, passing the `RT` object around is consistent with the intended meaning, which is just to pass an interface for looking up objects in imported modules.

The first step in implementing this scheme is to adapt the import context to take its arguments and return a value in the types of Java. The target type is the following:

```
type ImportContext_Java = JString -> JString -> IO JCommonObject
```

where the type `JString` represents a pointer to a string object of the JVM. The next step is to take it to a C function, which we do using the FFI.

Now that we have a C function properly typed for the JNI, we can pass a pointer to it to the constructor of the RT object.

```
RT(int import_context_ptr)
```

The native method `sym` of the RT class is declared in Java as follows:

```
public native CommonObject sym(String module_name, String symbol);
```

and implemented straightforwardly in C by casting the integer field to the proper function type and doing the call.

In the loader instantiates the `RT` object, thereby passing the wrapper function for the import context, and assign the `RT` object to the `rt` instance field of the `Module` object.

4.3 Haskell

Implementing support for Haskell in MLPE is much facilitated by the fact that it is MLPE's own implementation language. The task is far less involved than it was for C or Java in terms of low-level programming. On the other hand, Haskell is more disciplined

than the languages we used so far, and we have to adapt the way the system interfaces with module. In addition, we implement a scheme for exposing some services of the programming environment to the loaded Haskell modules.

4.3.1 The Haskell Compiler and Dynamic Loader

GHC is used to compile the modules written in Haskell, and the `SmartLinker` module [Ram], a dynamic loader for GHC, is used for loading the modules in MLPE. (There is no choice but to use GHC for compilation, because the dynamic loader expects the objects files to be in the format that GHC produces.)

4.3.2 Module Entry Point and Interface to MLPE

There are no global variables in Haskell, so we cannot rely on them to hold the import context and make it accessible to the module's functions, as we did in C. This suggests that the entry points for the module descriptor and for the import context should somehow be coupled.

Logically, the module describes itself to the runtime system and in return, the system provides the module with the import context. Programmatically, we would like to do the opposite: have the system pass the import context as a parameter to the function which provides the module description; that is, we would like to define the module entry point as a function of this signature:

```
describeModule :: ImportContext -> ModuleDescriptor
```

Of course, the import context is dependent on the module description, because the list of imported modules is taken from the module description. In spite of this fact, we can manage with a module entry point of the above type. Let us recall the type of an import context:

```
type ImportContext = ModuleName -> Name -> IO CommonObject
```

The fact that it is an IO function provides the necessary flexibility. The key is to construct a surrogate import context, which is programmed in terms of an access to a reference to another import context, as follows:

```
import_context =
      \ mod_name -> \ symbol ->
           do ctx <- readIORef import_context_ref
              ctx mod_name symbol
```

In the loader, the reference is first set to a dummy value (an empty import context). The loader passes the surrogate import context to the module's entry point function, and thus obtains the module descriptor; the loader can thus create and return the internal representation of the module. Subsequently, in the import context handler (which is passed the actual import context as argument,) the reference is set to the actual import context for the module, so that the surrogate import context becomes valid.

4.3.3 Exposing Services of MLPE

To expose particular services of the programming environment to the loaded modules, it suffices to do it in one language. This language can then be used to write a module that gives access to these services, and thus make them available to all supported languages. The distinguished language for doing this is the system's implementation language, Haskell.

To provide the modules with additional services, the signature of the module entry point is changed so that, instead of passing just the import context to the module, the system passes a compound structure which includes the import context as well as additional functions. The type `RuntimeInterface` is defined for this purpose, and the module entry point is defined as follows:

```
describeModule :: RuntimeInterface -> ModuleDescriptor
```

The service we wanted in our experiment was the access to the Haskell dynamic loader. This service adds a few functions to the interface:

```
data RuntimeInterface = RuntimeInterface {
    sym                          :: ImportContext,
    setLoaderObjectDirectory     :: String -> IO (),
    loadHaskellModule            :: String -> IO LoadedModule,
    unloadHaskellModule          :: LoadedModule -> IO (),
    loadHaskellFunction          :: forall a. LoadedModule -> String -> IO a
}
```

Chapter 5 The Language Prototyping Tool

For all practical purposes, in constructing a prototype interpreter, a concrete representation of programs must be supported. For this purpose MLPE provides a tool for performing some form of static program analysis. The tool covers lexical and syntactic analysis as well as basic typing. This chapter describes the design of this tool.

In Chapter 6, the prototyping tool is put to use to construct a prototype implementation of a simple language. Throughout the current chapter, we refer to particular elements of its implementation (see Appendix 2) to demonstrate particular features.

5.1 Design Goals

The prototyping tool is meant to serve as a basis for subsequent dynamic semantic analysis, which bears the real challenges. This tool should provide a basic facility that allows the user to easily express the concrete structure of an object language and provide a means to recover the structure of programs in a form that allows for convenient manipulation.

There are two aspects to the treatment of syntax and typing:

- specification: to capture a formal description of a particular aspect of a language, and
- analysis: resolving the basic structure of programs and other relevant properties based on some representation of the programs.

Specification and analysis have divergent requirements. In matters of specification, foremost requirements are correctness, clarity, conciseness, elegance and safety. A proper notation makes the system more useful. On the other hand, analysis matters are mostly driven by efficiency requirements.

5.2 Design Overview

The three components of the prototyping tool – which handle respectively lexical analysis, parsing, and type analysis – follow a common scheme. Each component defines

1. a type for specification,
2. a type for an analysis function, and
3. an "analyzer-generating" function: a function that returns an analysis function given the specifications.

For instance, in the case of lexical analysis, the prototyping tool defines a type for lexical specifications of a language (`Lexicon`), a type for lexical analyzer function (`Scanner = String -> [Token]`), and an analyzer-generating function of type `Lexicon -> Scanner`.

It is intended that the user, for the purpose of implementing some language, will write down the specification as values in the provided specification types and invoke the analyzer-generating functions to obtain an implementation. That is, the implementation of a language, say L1, would contain lines like these:

```
lexical_spec_L1 = Lexicon <...>
scanner_L1 = generate_scanner lexical_spec_L1
```

Another point that the three components have in common is the way in which the analyzer-generating functions work. In each of the three components, the analyzer-generating function returns an analysis function that is programmed in terms of some intermediate representation; this representation is distinct from the specifications, and facilitates analysis. For instance, the function that produces a scanner actually returns a scanner in the form of a function that is programmed in terms of some structure that contains the representation of a number of deterministic automata (this structure is of type `DFA_set`). In more concrete terms it means that, for instance, the scanner-generating function has this form:

```
generate_scanner :: Lexicon -> Scanner

generate_scanner lexicon =

   \input_string -> <clause in terms of dfa_set>    -- the returned scanner

where dfa_set = make_dfas lexicon        -- intermediate representation
```

The main types involved in the three components are shown in Figure 7. The bottom row shows the three specification types, the middle row shows the intermediate representations and the top row shows the types of the analysis functions in terms of their input and output types. As can be seen from the figure, the output type of lexical analysis coincides with the input type of parsing, and the output type of parsing coincides with the input type of type analysis, so that the generated analysis functions can be combined to form an analysis pipeline.

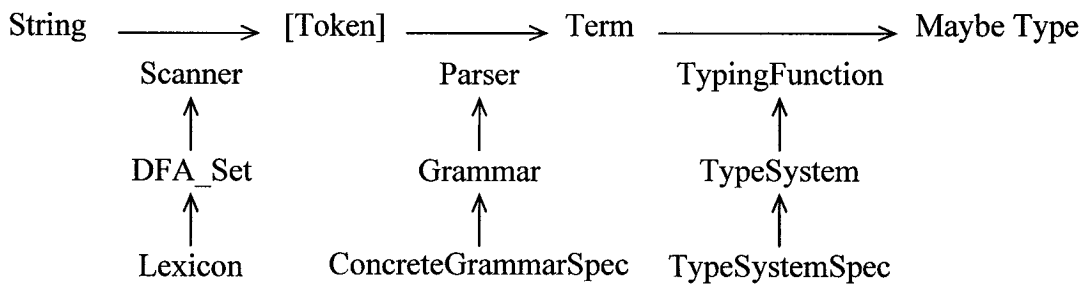

Figure 7. Main types of the language prototyping tool.

5.3 Lexical Structure

Lexical analysis is one of those amply studied problems having a basic, universally accepted solution. We are of course referring to basic automata theory. There is thus little interest in the analysis part, and attention will rather be given to specification issues. Our design guidelines will be put to use in this first, simple exercise.

5.3.1 Specification

The purpose of lexical analysis is to turn a stream of symbols from an alphabet into a stream of more meaningful symbols to be input to the later analysis phases.

It is conventional to view the lexical structure of a language as a set of distinguished lexical categories, each category being associated with a small regular language, which is

itself represented by a regular expression. A specification for a language will thus be made of such a set of *lexical rules*; a lexical rule defines a lexical category and the associated language; in addition, we include a regular expression for ignored input, which is normally made of blank characters, line breaks and comments.

We use three types for lexical specifications:

1. one for the lexical structure of an entire language (`Lexicon`),
2. one for a single entry in such a specification (`LexicalRuleSpec`), and
3. one for regular expressions (`RegExp`).

```
data Lexicon = Lexicon [LexicalRuleSpec]      -- individual rules
                       (RegExp Char)          -- ignored input
data LexicalRuleSpec = LRS Terminal (RegExp Char)
```

Regular expressions are formally made of symbols from an alphabet (a), unions of regular expressions $(s|t)$, concatenations (st), indefinite repetitions (s^*), the empty string (λ) and the empty language ($\varnothing$) [Linz01]. They are modeled slightly differently here. Instead of individual characters, *sets* of characters are modeled – in this way, cumbersome unions that would result in unnecessarily large and inefficient automata are avoided. Note that the empty language is useless in practice, but it can still be modeled it using an empty set of characters.

```
data (Ord sym)
     => RegExp sym =
          RE_Set      (Set_ sym)
        | RE_Lambda
        | RE_Union    (RegExp sym) (RegExp sym)
        | RE_Cat      (RegExp sym) (RegExp sym)
        | RE_Star     (RegExp sym)
```

In addition to the constructor function, the tool provides helper functions to create common patterns:

```
re_Plus   :: RegExp -> RegExp

re_String :: String -> RegExp
```

The first function creates the "plus-closure" of a regular expression, meaning indefinite repetition with at least one occurrence. The second creates a regular expression for a given string using a concatenation of singleton sets.

It would be convenient and fairly conventional to work from a textual representation of regular expressions, but we have managed without it so far.

An example of a lexical specification can be found in Appendix 2, where it is bound to the name `v_lexicon`.

5.3.2 Analysis

Let us first define the interface to the scanner. Taking advantage of the laziness of Haskell, we can simply use the type

```
type Scanner = String -> [Token]
```

and get the intended stream behavior without any need for explicit synchronization. Left to choose is the token representation. The representation could distinguish between those tokens which have a material lexeme and those which don't, but for the sake of simplicity, the lexeme is held for all tokens. The information stored about a token is a terminal symbol (identifying the lexical category) and the lexeme:

```
data Token = TOKEN Terminal Lexeme
```

The representation used for analysis is elaborated in this way: Each regular expression is taken to a nondeterministic finite acceptor (NFA). NFA's are cumbersome and slow to trace, so each NFA is converted to an equivalent deterministic finite acceptor (DFA), which can be traced much faster. The conversion to DFA is done using the subset construction [Linz01].

5.4 Syntax

The treatment syntax is based on three central representations: that of

- the abstract syntax of a language,

- the concrete syntax of a language, and
- terms in the languages.

In brief, the prototyping tool defines a uniform representation of terms; the abstract syntax defines the form of the valid terms; and the concrete syntax defines how sequences of tokens can be assembled into terms. When writing an interpreter, the user writes down the specification of the abstract and concrete syntax of the language; a parsing function is then obtained with a call to the analyzer-generating function. Semantic functions are defined over the uniform representation of the terms. An example is the implementation of the V language shown in Appendix 2.

In practice, the specification of the abstract syntax can be omitted (as is done in Appendix 2.) The specification of the abstract syntax could be used to check the concrete syntax for correctness (i.e. to assert that the parser can only produce valid terms) but this verification is not implemented.

5.4.1 Specification of Abstract Syntax

Here the requirement for full declarativity is somewhat less trivial to meet. The bottom line to be able to construct terms in some abstract language, exempt from the peculiarities of the concrete syntax – this should be a simple structure that is convenient to analyze and manipulate. Let us first define this representation.

Suppose we are writing a simple parser in Haskell; to be concrete, suppose it's a parser for a simple language for doing arithmetic over constants and variables. We might start with a type like the following:

```
data Term = Constant Int              -- n
          | Variable String           -- x
          | Plus Term Term            -- s + t
          | Times Term Term           -- s * t
```

This type would normally be made the return type of the parser. In the prototyping tool, in order to be able to do analysis over the type itself, this syntax must be represented as a first class value, so a type to express the abstract syntax of an object language is needed.

In the above abstract syntax, three types are involved: `Term`, `Int` and `String`. In formal syntax terminology, one would rather speak of *carriers* instead of types, and we will henceforth do so. `Term` and the other two carriers are clearly of a different nature: `Term` has a structure defined by the actual syntax, while the other two represent primitive data. As these two kinds of carriers have different features, they are modeled distinctly. We call *ground carriers* those carriers who carry primitive data, and *ordinary carriers* those carriers whose structure is defined by the actual syntax.

For all practical purposes, ground carriers hold primitive data returned by the scanner (in particular, the lexemes.) A uniform representation of this data will suffice: it can be represented uniformly as strings. Thus, ground carriers basically have no feature other than identity that need be specified. Ordinary carriers have features that need specification, and they will be defined using a separate type.

The type used for specifying an abstract syntax is the following:

```
data AbstractGrammarSpec =

    AbstractGrammarSpec [CarrierName] -- Names of the ground carriers
                        [CarrierSpec] -- Specs the new carriers
```

In the specification of individual ordinary carriers, the carriers are equipped with a set of *constructors*. A constructor is a function of given arity, taking arguments from definite carriers, and injecting terms back into some definite carrier. Constructors also have a name. The argument carriers may be ground carriers or ordinary carriers defined in the same specification. The following type is defined for specifying ordinary carriers and their constructors:

```
data CarrierSpec = CRS CarrierName [ConstructorSpec]

data ConstructorSpec = CNS ConstructorName [CarrierName]
```

Continuing with the example of the simple language, its abstract syntax would be specified as follows:

```
abstract_syntax =

   AbstractGrammarSpec

      -- ground carriers:

      ["Int", "String"]

      -- ordinary carriers:

      [CRS "Term"  [CNS "Constant"   ["Int"],

                    CNS "Variable"   ["String"],

                    CNS "Plus"       ["Term", "Term"],

                    CNS "Times"      ["Term", "Term"]]]
```

Next, a representation of terms in such syntax must be defined. For terms in ground carriers, the carrier must be identified and the lexeme provided. We call such terms *atoms* to distinguish them from structured terms. For terms from ordinary carriers, the constructor is identified long with the arguments. The following type is used:

```
data Term

     = Atom CarrierName Lexeme

     | Term ConstructorName [Term]
```

For illustration, a term like $3x + 2$ would be represented as

```
   Term "Plus" [Term "Times" [Term "Constant" [Atom "Int" "3"],
                                    Term "Variable" [Atom "String" "x"]],
                    Term "Constant" [Atom "Int" "2"]]
```

5.4.2 Specification of Concrete Syntax

The prototyping tool performs LALR parsing [Parsons92], which has amply proven to be a viable compromise between generality and efficiency.

Using code from the existing parser generator Happy [Happy] we have assembled a custom parser generator to suit our requirements. Happy is an LALR parser generator for Haskell in the tradition of Yacc. While the intricate part of constructing parsing tables was taken as is, major modifications to the overall organization of the parser-

generator were needed; in particular, the interface to the parser generator had to be rearranged and a generic parser driver had to be written.

The input to Happy is a specification file containing production rules along with functional code, much in the manner of Yacc but with a Haskell flavor. This is clearly unhandy in the current context – instead, data types that can be used for specification should be provided, so that the input grammar can fit in a Haskell file as well as the other components of a language's specification.

A grammar specification defines a set of non-terminal symbols and, for each symbol, a set of production rules. It also associates each non-terminal symbol with a carrier in the abstract grammar: all terms for a given non-terminal are mapped into the same carrier. A production rule lists tokens and non-terminal symbols, and identifies a constructor from the abstract grammar. The numbers in brackets (<1>, <2>...) indicate the position of the argument to be passed to the constructor. This is a declarative (and elementary) approach – in this way, only manipulate primitive data is manipulated. It has the advantage that it can be checked for safety. An example of a specification of a concrete grammar can be found in Appendix 2, where it is bound to the name `v_concrete_grammar`.

5.4.3 Analysis

Happy generates parser code in the form of a series of Haskell functions, in such a way that the parsing tables do not appear in the output. (Table lookup is actually expressed using pattern matching in Haskell functions.) Such output is useless to us in the current context, so we need to recourse to Happy's internal tables and write our own parsing function.

It is convenient and logical to divide the parsing process in two phases:

1. parsing itself, where a representation of terms in the concrete grammar is created; this representation refers to the actual productions in Happy's production list, and

2. term assembly, where the above representation is analyzed and related to our concrete grammar in order to construct terms in the abstract syntax.

The inner function of the synthesized parser is a transition function which takes a parse one step ahead by doing either a shift or a reduce action. The function is of the following type.

```
type ParserTransitionFunction =

   ParserContext -> (ParserStack, [Token]) -> (ParserStack, [Token])
```

All constant information describing the parser is contained in a value of type `ParserContext`. The parser itself applies the transition function repeatedly, passing the new stack and input every time, and monitoring the stack and input for acceptance or rejection which are the stop conditions. The first pass yields parse trees of this type:

```
data ParseTree

     = PT'Token      Terminal Lexeme

     | PT'Compound  Name      [ParseTree]
```

where the `Name` field identifies a production in the concrete grammar. The second pass constructs terms in the abstract grammar by looking up the identified productions in the concrete grammar and constructing a term accordingly, in a compositional manner.

5.5 Typing

It is not clear whether simple static type analysis, as presented it in this section, can actually be useful in writing a prototype implementation of a language that is at all realistic. The tropic is deemed relevant to this thesis though, be it at least in its quality of an additional example of our stated design guidelines and a logical next step after syntactic analysis.

5.5.1 Specification

The specifications of type systems cast traditional descriptions of type systems, as described in [Cardelli97], in the style of specification that we have been using so far. The specification of a type system is a construction built on top of an object language represented by its abstract grammar. It associates a type to some of the terms and leaves other terms untyped.

It is assumed that the object grammar defines a carrier named `Type`, whose members are all valid types; the inferred types are represented as terms from this carrier.

The specification of a type system consists of a set of typing rules. A typing rule specifies a set of hypothesis judgments and a goal judgment. A typing judgment is a statement that a certain term in a certain environment has a certain type. Because the environments, terms, and types appearing in typing judgments may contain meta-variables, we refer to them as *meta-environments*, *meta-terms* and *meta-types*, respectively.

The use of the type checking facility is demonstrated below using the well-known example of the simply typed λ-calculus; in order to make its typing more interesting, the language is extended with integer arithmetic and pairs. The abstract grammar is expressed in conventional notation in Figure 8, and the type system in Figure 9.

$T ::=$	terms
x	variable
n	integer constant
$\lambda x{:}T.t$	function abstraction
$t\ t$	function application
(t, t)	product construction
$fst\ t, snd\ t$	product elimination
$t + t, t - t, t \cdot t, t / t$	integer arithmetic
$T ::=$	types
Int	integer type
$T \times T$	product types
$T \rightarrow T$	function types

Figure 8. Abstract syntax of the simple functional language

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T}{\Gamma \vdash (s, t) : S \times T} \qquad \frac{\Gamma \vdash t : (S \times T)}{\Gamma \vdash fst\ t : S} \qquad \frac{\Gamma \vdash t : S \times T}{\Gamma \vdash snd\ t : T}$$

$$\frac{\Gamma[x : S] \vdash t : T}{\Gamma \vdash \lambda x : S.t : S \rightarrow T} \qquad \frac{\Gamma \vdash s : S \rightarrow T \quad \Gamma \vdash t : S}{\Gamma \vdash s\ t : T}$$

Figure 9. Typing rules for products and functions

The abstract syntax is expressed using our specification formalism as follows:

```
AbstractGrammarSpec

    -- ground carriers

    ["INT", "VAR"]

    -- ordinary carriers

    [CRS "Term"

           [CNS "var"     "VAR",            -- x

            CNS "int"     "INT",            -- n

            CNS "abs"     "VAR Type Term",  -- \x:T.t

            CNS "app"     "Term Term",      -- t t

            CNS "pair"    "Term Term",      -- (t, t)

            CNS "pls"     "Term Term",      -- t + t (and similarly -,*,/)

            CNS "fst"     "Term",           -- fst t

            CNS "snd"     "Term"       ]    -- snd t

     CRS "Type"

          [CNS "Int"     "",               -- Int

           CNS "Prod"    "Type Type",      -- (T, T)

           CNS "Func"    "Type Type"  ]]   -- T -> T
```

The specification of the type system is as follows.

```
TypeSystemSpec
    -- object (abstract) grammar
    abstract_grammar_spec
    -- type rules
    [-- product construction
     TRS [TJS "$E" "$x" "$S", TJS "$E" "$y" "$T"]
         (TJS "$E" "pair $x $y" "Prod $S $T"),
     -- product elimination
     TRS [TJS "$E" "$x" "Prod $S $T"]
         (TJS "$E" "fst $x" "$S"),
     TRS [TJS "$E" "$x" "Prod $S $T"]
         (TJS "$E" "snd $x" "$T"),
     -- lambda abstraction
     TRS [TJS "$E[$v:$V]" "$t" "$T"]
         (TJS "$E" "abs $v $V $t" "Func $V $T"),
     -- function application
     TRS [TJS "$E" "$f" "Func $A $B", TJS "$E" "$x" "$A"]
         (TJS "$e" "app $f $x" "$B")]
```

5.5.2 Analysis

Given the specification of a type system, the tool derives a more useful representation which is simply a mapping from constructors in the abstract grammar to sets of typing rules. This data structure is meant to be used by the typing function itself. The typing function has the following type:

```
type TypingFunction
   = TypingEnvironment -> Term -> Maybe Type
```

In trying to infer a type for a term in a given environment, the typing function first identifies which typing rules potentially apply. This is done by trying to unify the meta-

term in the goal with the input term. If it succeeds, the unification procedure returns an initial set of bindings.

For every hypothesis, the initial bindings are applied to the meta-environment to yield a new environment which reflects a potential environment extension. The bindings are then applied to the meta-term, which yields a term with no free meta-variables. The typing function is then applied to this new term, and if it is found to be well typed, then the resulting type is unified with the meta-type in the hypothesis. If this process succeeds for every hypothesis and produces a set of binding which is consistent, then the resulting bindings for every hypothesis are merged into a single set of bindings which is finally applied to the meta-type; the result of this last substitution is the value returned by the typing function.

Let's take the example of lambda abstraction, say the identity function on integers, $\lambda x{:}Int.x$, for which we try to infer a type in the empty environment. The term is represented in the abstract grammar as `Term "abs" [Atom "var" "x", Term "Int" [], Term "var" [Atom "var" "x"]]`. The only potential typing rule is the following.

```
TRS [TJS "$E[$v:$V]" "$t" "$T"]
    (TJS "$E" "abs $v $V $t" "Func $V $T")
```

The initial unification succeeds, binding `Atom "var" "x"` to `$v`, `Term "Int" []` to `$V` and `Term "var" [Atom "var" "x"]` to `$t`. The empty environment would be extended to `[$v:Term "Int" []]`. The system would apply the initial binding to the meta-term, `$t`, yielding `Term "var" [Atom "var" "x"]]`. The typing function would be applied to this term, in the extended environment, yielding the type `Term "Int"`. Unification with the meta-type in the hypothesis yields the additional binding of `Term "Int" []` to `$t`. Applying the combined bindings to the meta-type in the goal judgment, the resulting type, expressed in the abstract grammar, is `Term "Func" [Term "Int" [], Term "Int" []]`, which effectively represents the expected type, $Int \rightarrow Int$.

5.6 Summary

We have developed an integrated tool for static program analysis where the specification of the static features of a language can be expressed as Haskell values, so that it

facilitates the construction prototype interpreters for their use in MLPE without any need to use external tools. The user needs only define the specifications – of the lexicon, syntax, and possibly the typing rules – and invoke the analyzer-generating functions to obtain an implementation.

Chapter 6 A Prototype Language Implementation

In this chapter, a simple language is designed and implemented using the prototyping tool, and its implementation is integrated into MLPE. It enables one to appreciate the suitability of the prototyping tool as an aid to language design, as well as the suitability of the programming environment for hosting a prototype implementation.

The task of integrating a prototype implementation into the environment ought to be, in many respects, similar to that of integrating a black-box implementation – alike, it involves implementing marshaling functions, entry points, module descriptors, etc. The design parameters, however, differ significantly. When using a black-box implementation, the design of the loader and support function is constrained by the peculiarities of the language and its implementation. When designing a language and constructing a prototype implementation, more design elements are at hand.

The semantic modeling used in this chapter is purposely simplistic. The example language is developed in the simplest possible semantic framework, so as to be better able concentrate on the issues related to integration, which are more relevant to the subject matter of this thesis.

6.1 The V Language

The language designed in this section has the admittedly contrived purpose of exposing the features of MLPE, and in particular its type system. We call it "V", because it is meant to provide a simple means to define and manipulate values in the common type system. The intrinsic qualities of this language are of little interest here, as long as the language suffices for the experiment. The intent is to design a simple language with a straightforward implementation so as to ease the presentation.

6.2 Semantics and Implementation

V is a simple imperative language that supports the definition and invocation of procedures, as well as explicit assignments and I/O operations. The primary objects of the language are the values in the common type system.

A logical place to start in the design of the language is to identify the syntactic categories, the semantic domains and the types of the semantic functions. In order to have the simplest possible semantic functions, the state is modeled as a direct mapping from symbols to values. The semantic functions will be significantly simpler than if a separate environment and store were used. The following representation of states is used:

```
type State = Map_ String V
```

where V is the domain of first-class values. Two syntactic categories are defined: one for expressions and one for commands. The evaluation of both expressions and commands may have side effects. Clearly, effects on the state must be modeled; however, other side effects such as IO or other calls are not relevant to this semantics, and are not modeled in the semantics. Instead, the semantic functions themselves are made IO functions, so that evaluating commands or expressions may have arbitrary side effects that are not accounted for.

Informally, the semantics of an expression is that, when evaluated in a given state, it returns a value, it may alter the state and it may also have other side effects not accounted for. The semantic function is typed as follows:

```
semV :: Exp -> State -> IO (V, State)
```

The semantics function for commands is similar but simpler in that it does not return a value:

```
semC :: Cmd -> State -> IO State
```

As to the semantic values themselves, they basically mirror the common type system. To represent primitive and structured data from the common type system, the representation of V values includes a set of constructors:

```
-- values

data V  = V'N Int                       -- integer
        | V'S String                    -- string
        | V'P V V                       -- pair
        | V'L [V]                       -- list
        | V'V                           -- void
        ...
```

Procedures are modeled as values as well. Procedures are modeled in V more simply than in the common type system; in particular, the distinction between pure functions and functions with side effects in not accounted for: all procedures are allowed to have side effects. For the purpose of exporting procedures to the common type system, the type of the parameters and return values must be known. In consequence, both an explicit representation of types, and a type argument in the constructor for function objects are needed. The constructor for procedures is defined as follows:

```
data V  = ...
        | V'F T (State -> [V] -> IO (State, V))  -- procedure
          ...
```

where the type argument, of type `T`, defines the function signature; it is intended to be an instance of the following constructor:

```
data T  = ...
        | T'Func [T] T
          ...
```

The first argument to the constructor enumerates the types of the procedure arguments, and the second argument identifies the return type.

The full definition is of the representation of types is defined trivially as follows:

```
data T  = T'Int
        | T'String
        | T'Pair T T
        | T'List T
        | T'Func [T] T
        | T'Void
```

In the lack of environments or frames in our semantic modeling, the treatment of procedures has to be a somewhat simplistic, but a simple mechanism for defining procedures that will suffice for our experiment can be devised, as follows. For parameter passing, the parameters are used as ordinary variables – calling a function with a given set of arguments is equivalent to doing a number of assignments and then executing the procedure body. For the return value, a "return" statement is provided, whose effect is to set the variable named `ret` to the supplied value. Note that this is not the usual semantics of "return", as the semantics defines no effect on control. However simplistic, this modeling of procedures is sufficient to define recursive functions, as demonstrated in Section 7.7 with the factorial function.

Finally, syntax is included for expressing integers, strings, pairs and lists. In addition, a number of built-in procedures for doing I/O and extracting components of compound structures are included in the form of ordinary V procedures (these procedures are loaded into an initial state.)

So much about the design of V. Given the basic characteristics of the language, the remaining tasks of implementation bear little difficulty: it is straightforward at this point to write down the actual concrete syntax and the full implementation of the semantic functions. The full code is given in Appendix 2.

6.3 Integration into MLPE

For the purpose of interfacing with MLPE, V is augmented with specialized syntax. In particular, additional syntax is provided for

- stating the module's name,

- exporting symbols,
- importing modules, and
- referring to symbols from other modules.

Haskell-like syntax is adopted for the above items. For the first two items, syntax is provided for defining a *module header*, where the module name is indicated, the exported symbols are enumerated in parentheses, and the body of the program is introduced in a "where" clause. The actual syntax rule is as follows:

```
RLS "module VAR<1> ( Elist<2> ) where Cmd'<3>" "module"
```

where `Elist` stands for a comma-separated list of exported symbols. This rule actually defines a new command whose effect if to define a module. Semantics must be assigned to this command in such a way that the loader can gather the module name and exported symbols by evaluating the command. In the current semantic framework, the designated way to pass such information is through the state. The module name is stored in a variable of predefined name in the state; similarly, the list of exported symbols, represented as a list of strings, is stored in a variable of predefined name.

For importing modules, a separate "import" statement is defined:

```
RLS "import VAR<1>"                    "import",
```

Its effect is to append the specified module name to a list of imported modules that is stored in a variable of predefined name in the state.

Finally, for modules to access symbols defined in other modules, a foreign symbol resolution operator with the customary "dot-notation":

```
RLS "VAR<1> . VAR<2>"                   "."],
```

The semantics of this expression is to invoke a procedure that is assumed to be bound to the name `import_context` in the state, and that is assumed to take two string arguments, the name of the module and the requested symbol, and to return the corresponding object – this procedure is assumed to be installed into the state by the loader.

6.3.1 Marshaling Functions

The marshaling functions for V are much easier to realize that those for C or Java, because values in V are represented using a Haskell types. Since the marshaling functions do not rely on constructors or inspector functions from foreign languages, they can be implemented as pure functions rather than IO functions.

The marshaling functions over types are especially straightforward. Their types are as follows:

```
marshalType   :: CommonType -> T
unmarshalType :: T -> CommonType
```

There is some complication with the marshaling of values, however. As usual, the issue comes with marshaling functions. In order to apply a procedure defined in V, the state must be supplied to the semantic function. Therefore, in order to do invocations, the state has to somehow pass through the marshaling functions. However, as a matter of design, it is clearly not a responsibility of the marshaling functions to manage state updates. It is more desirable to have an third peer that sits between the semantic functions and the marshaling functions and tracks state updates. For this purpose, the type `V_Inst` is introduced to represent a loaded module (and track its state.) The type is described in the next section.

Returning to the marshaling functions, their types are as follows:

```
marshalObject   :: V_Inst -> CommonObject -> V
unmarshalObject :: V_Inst -> V -> CommonObject
```

For exporting functions of the common type system to V, the marshaling function casts the function value to a V function having no effect on the state.

For importing functions from V, the partial application issue must be resolved. To this end, yet another variant of the "defer" function is implemented. When applying the V function, the state is obtained from the `V_Inst` value; after the call, the new state yielded by the function application is committed back to the `V_Inst` value.

6.3.2 Module Instance Wrapper

The type `V_Inst` characterizes an instance of a loaded V module. A value in this type holds a reference to the current state:

```
type V_Inst = IORef State
```

For the use of the loader, functions are provided to create the module instance given the program text, as well as functions to retrieve and update the state:

```
v_init   :: String -> State -> IO V_Inst
v_state  :: V_Inst -> IO State
v_update :: V_Inst -> State -> IO ()
```

Finally, we recall that the main reason for which we write this wrapper is the need to pipe the state through the marshaling function. The state is actually needed in only one case: when importing a function from V, a state must be supplied in order to apply the function. For this purpose, one last function is provided, which takes as argument a V function and its parameters; it applies the function, commits the new state to the module instance, and returns the value that the function has returned.

```
v_apply_proc :: V_Inst -> V -> [V] -> IO V
```

6.3.3 Loader

The V loader, unlike those of Chapter 4, operates directly over the source code.

The loader function first loads the program text and creates the `V_Inst` object by passing the program text and an empty state to the initialization function of the module wrapper. Then, using the same technique used previously with Haskell modules, it instantiates an import context in the semantic domain of V – it is done by first creating a reference to an empty import context, and programming the lookup function in terms of that reference. The symbol lookup function is then added to the state. Next, the loader collects from the state the names of the imported modules, as well as the exported symbols and their individual values, which are fetched from variables of the same name in the state. With this information, the `Module` object is created and returned. Finally, in the body of the import context handler, the reference to the import context to the actual import context

for the module, so that symbol lookups effectively give access to symbols defined in other modules.

Chapter 7 Sample Project

In this chapter a simple project developed using MLPE is presented. The project has the contrived purpose of demonstrating the capabilities of MLPE, and does not otherwise fulfil a particular purpose.

An overview of the project is given first. The specification pf the project, the directory structure, and the individual modules are presented in turn. Along with the presentation of the individual modules, samples sessions in the MLPE shell are shown.

7.1 Project Overview

The project, named `P1`, is composed of four modules, named `C1`, `J1`, `H1` and `V1`, that are implemented respectively in C, Java, Haskell and our prototype language, V. Each module aims to demonstrate the support of the individual languages presented in Chapter Chapter 4 and Chapter 6.

Each module defines a number of values and functions. Each module also makes use of the services of another module; more precisely:

- `C1` imports `J1`,
- `J1` imports `H1`,
- `H1` imports `V1` and, (making a cycle,)
- `V1` imports `C1`.

As an aside, this structure illustrates the fact that cyclic dependencies among modules do not cause problems in MLPE.

7.2 Project Specifications

The project specification (Section 3.4) file is shown in Appendix 4.

The specification is written in a Haskell module that is dynamically loaded by MLPE on start-up. This module imports the `Project` module, which defines the type for project specification. The module also exports the symbol `spec`, which is the predefined entry point for project specification modules.

The project specification itself (the `spec` value) lists the general properties of the project along with the specification of the individual constituents of the project. The general properties are the name of the project, the project home directory and the default target module, which is the module that MLPE loads on start-up.

The interpretation of the specification of the project constituents is the following (it reads bottom-up in the specification):

- the module `C1` is loaded from the dynamic library `C1.so`, which is itself linked from the object file `C1.o`, whose originating source file is `C1.c`;
- the module `J1` is loaded from the class file `J1.class`, whose originating source file is `J1.java`;
- the module `H1` is loaded from the compiled Haskell module `H1.o`, whose originating source file is `H1.hs`;
- the module `V1` is loaded directly from the source file `V1.v`.

7.3 Directory Structure

The simple conventions used in MLPE regarding the disposition of the project's files are defined in Section 3.4.1. In the current project, the files are laid on the file system as follows:

```
./                          -- project home directory
   C1.c                     -- source files
   J1.java
   H1.hs
   V1.v
   obj/                     -- resource directory
     C1.o
     C1.so
     J1.class
     H1.o
   spec/                    -- specification directory
     ProjectSpec.hs
     ProjectSpec.o
```

In particular, it is intended that MLPE be invoked from the project home directory, so that the location of project specification file needs not be specified on the command line.

7.4 The C Module

Let us begin with a quick look at how the module presents itself in the MLPE shell. The following is a short session in which the user requests to display the module's contents:

```
$ mlpe

                 ___    ___
   |\  /| |     |    \ |
   | \/ | |     |___/ |__
   |    | |     |     |     Multi-Language Programming Environment
   |    | |___  |     |___  Version 0.5
loading module C1.
loading module J1.
loading module H1.
loading module V1.
mlpe# :m C1
Module C1 [J1]
  type Counter
  double = {Int -> Int}
  getPWD = {IO String}
  getValue = {Counter -> IO Int}
  hello = {IO Void}
  increment = {Counter -> IO Void}
  l = ["a", "b", "c"]
  map = {(Void -> Void) -> [Void] -> [Void]}
  n = 100
  newCounter = {IO Counter}
  p = ("Left", "Right")
  reset = {Counter -> IO Void}
  s = "<some C-string>"
  swap = {(Void, Void) -> (Void, Void)}
  timesNine = {Int -> Int}
mlpe#
```

The name in brackets (`J1`) indicates the module `C1` imports the module `J1`. The remainder of the output list the individual symbols (referring to values and functions) defined in the module.

The user can evaluate simple expressions in the shell by typing them at the prompt. The user can also initiate a rebuild, which automatically recompiles the files that have changed on disk and reloads the modules. Supposing that the user changes the value of the symbol `n` in the source code from `100` to `1000`, the effect can be readily seen in the shell as follows:

```
mlpe# C1.n
  ==> 100
mlpe# :r
                 ___   ___
   |\  /| |     |   \ |
   | \/ | |     |___/ |__
   |    | |     |     |     Multi-Language Programming Environment
   |    | |___  |     |___  Version 0.5
gcc -c C1.c -I/home/louis/mlpe/latest/include/ -o obj/C1.o
gcc -shared obj/C1.o -o obj/C1.so
loading module C1.
loading module J1.
loading module H1.
loading module V1.
mlpe# C1.n
  ==> 1000
```

The source file of the module `C1` is listed in Appendix 5.

The first line includes the file `mlpe_runtime.h`, which defines the function signatures for interfacing with MLPE. The second line defines the `rt` variable, which is used by the functions of the module to access values and functions defined in other modules.

The function `describeModule` (at the bottom of the source listing) is the module header – it is called by MLPE when the module is loaded to obtain the module definition. The module header indicates the name of the module, the modules to be imported, the names of any opaque types defined in the module, along with the symbols (functions and values) defined in the module.

A number of simple values are defined in the module `C1` – the values `n`, `s`, `p` and `l`, which represent an integer, a string, a pair and a list, respectively. The other symbols of the module define functions. The constructors for functions and procedures take as argument a representation of the types of their domain and range; the constructor also takes as argument a pointer to the implementation of the functions.

The implementation of the individual functions is provided by the other functions defined in the source file. For convenience, the type of these functions is shown in Haskell-like syntax in comments. All arguments and return values to functions are of type `CommonObject*`, which is the representation in C of values in the common type system (see Section 3.2). Values of this type are manipulated in C through a number of accessor and constructor functions (defined in Section 4.1.2.2). For instance, the function `coInt()` creates an integer value in the common type system, and the `coGetInt()` function returns the integer value in C of an integer value in the common type system.

The first five functions defined in the module do basic data manipulation or produce simple effects:

- the `double` function doubles its integer argument;
- the `swap` function swaps the components of a pair;
- the `getPWD` function returns the current working directory;
- the `map` function, as in Haskell, applies a given function to every component of a list;
- the `hello` function displays "Hello, world!".

The function `map` differs from the other four in that it is a polymorphic (and higher-order) function. In Haskell syntax, its type would be expressed as `(a, b) -> [a] -> [b]`.

The common type system does not have type variables, however. We use the simple convention that the type `Void` can represent a polymorphic type argument. Thus, we represent the type of map as `(Void -> Void) -> [Void] -> [Void]`.

The user can invoke the above functions in the shell as follows:

```
mlpe# C1.double 10
  ==> 20
mlpe# C1.swap ("a", "b")
  ==> ("b", "a")
mlpe# C1.getPWD
  ==> "/home/louis/mlpe/0.5/projects/P1"
mlpe# C1.hello
Hello, world!
  ==> Void
mlpe# C1.map C1.double [1,2,3,4]
  ==> [2, 4, 6, 8]
```

The following four functions operate over the opaque type `Counter` (which is declared in the module header):

`newCounter` instantiates a counter;

`increment` increments a counter;

`reset` resets a counter to zero;

`getValue` retrieves the current value of a counter.

The following shell session fragment illustrates the use of the above four functions.

```
mlpe# c <- C1.newCounter
c = {Counter}
mlpe# C1.getValue c
  ==> 0
mlpe# C1.increment c
  ==> Void
mlpe# C1.getValue c
  ==> 1
mlpe# C1.increment c
  ==> Void
mlpe# C1.getValue c
  ==> 2
mlpe# C1.reset c
  ==> Void
mlpe# C1.getValue c
  ==> 0
```

MLPE defines a representation of values of opaque types that wraps a "void" pointer. Functions that manipulate such values cast the "void" pointer to the actual type of the opaque object. In the case of the `Counter` type, the state of a counter is represented as an integer value, so a `Counter` instance is represented as a pointer to an integer.

Finally, the `timesNine` function demonstrates the use of a function from an imported module. It multiplies its argument by nine by applying the `triple` function from module `J1` to its argument twice. It obtains a representation of the `triple` function by calling the `rt` function. Subsequently, it does a type cast to obtain a C function pointer properly typed as a function of one argument.

To complete the demonstration:

```
mlpe# C1.timesNine 5
  ==> 45
```

7.5 The Java Module

The source code of the module `J1` is shown in Appendix 6.

The module `J1` is implemented in a Java class of the same name. MLPE creates an instance of this class when loading the module, and fetches the module definition by calling the `describeModule()` method.

In Java, values in the common type system are represented as instances of a number of classes, which are provided by the package `rt`. These classes provide constructors and accessor methods to manipulate the data. Functions are defined by instantiating an object of class `CO_Func` (or `CO_IO` for a parameter-less procedure) and overriding its `func()` method with the function implementation.

In the source code, a number of simple data elements are defined directly in the `describeModule()` method; a number of functions are defined separately as class members.

The module implements the `swap`, `hello` and `map` functions with the same behaviour as those with the same name in module `C1`. In addition, it implements three simple functions:

- the `triple` function triples its integer argument;
- the `getClassPath` function returns the JVM's current class path;
- the `fourthPower` function computes the fourth power of an integer.
- The `fourthPower` function makes use of the `square` function from the module `H1`, which it applies twice. A representation of the `square` function is obtained by calling the `sym()` method of the `rt` class member, which is inherited from the `Module` class. The function can then be called by simply invoking the `func()` method over the function's representation.

To demonstrate:

```
mlpe# J1.map J1.triple [1,2,3,4]

  ==> [3, 6, 9, 12]

mlpe# J1.getClassPath

  ==>
"/pkg/jdk1.3.1/jre/lib/rt.jar:/pkg/jdk1.3.1/jre/lib/i18n.jar:/pkg/jdk1.3.1
/jre/lib/sunrsasign.jar:/pkg/jdk1.3.1/jre/classes:/pkg/jdk1.3.1/jre/lib/rt
.jar:/home/louis/mlpe/latest//java:obj/"

mlpe# J1.fourthPower 10

  ==> 10000
```

Finally, the module defines an opaque type named `Toggle`. In Java, opaque objects are represented as instance of the `Object` class. The represented object is recovered using a class cast. In the case of the `Toggle` opaque type, objects are represented using the `Toggle` inner class of the class `J1`. The module defines some functions to manipulate `Toggle` objects:

- `newToggle` instantiates a `Toggle` object;
- `toggle` toggles a `Toggle` object's state;
- `getState` retrieves a `Toggle` object's state,

the state being either zero or one. To demonstrate:

```
mlpe# t <- J1.newToggle

t = {Toggle}

mlpe# J1.getState t

  ==> 0

mlpe# J1.toggle t

  ==> Void

mlpe# J1.getState t

  ==> 1

mlpe# J1.toggle t

  ==> Void

mlpe# J1.getState t

  ==> 0
```

7.6 The Haskell Module

The source code of module `H1` is shown in Appendix 7.

The module `H1` is defined in a Haskell module of the same name. MLPE fetches the module definition from the symbol `describeModule`, which must be exported in the Haskell module.

The module imports the `HS_Runtime` module, which provides the necessary types and functions for implementing modules in Haskell. It particular, it gives access to MLPE's representation of the common type system (Section 3.2.2). It also provides a type for describing a module (`ModuleDescriptor`) and for interfacing with MLPE (the type `RuntimeInterface`.) Finally, it provides a number of helper functions to facilitate the implementation of functions in Haskell.

The `describeModule` function receives as argument a value of type `RuntimeInterface`, which contains a function to resolve symbols from imported modules (the function `sym`). The `RuntimeInterface` value also contains functions for accessing other services of MLPE that are not demonstrated here. The `describeModule` function returns the module description in the type `ModuleDescriptor`.

The `H1` module defines a number of primitive values as well as a number of functions. It provides a Haskell version of the functions `swap`, `hello` and `map`. In addition, it provides

- the `square` function squares its argument;
- the `getYear` function returns the current year;
- the function `C(n, r)` computes the number of `r`-combinations of a set of `n` elements.

To demonstrate:

```
mlpe# H1.map H1.square [1,2,3,4,5]
  ==> [1, 4, 9, 16, 25]
mlpe# H1.getYear
  ==> 2004
mlpe# H1.C 4 2
  ==> 6
```

The functions are implemented with the help of the `foldfunc` function. This function handles the "folding" of a multiple-argument function into MLPE's representation of functions, in which multiple-argument functions are modeled as nested one-argument functions (that is, functions returning functions.) The first argument to `foldfunc` is a list of the types of the arguments and return value of the represented function (the last component of the list being the return type.) The second argument is the function itself, which takes its arguments in a list. In the source code, the second argument is written as a lambda-abstraction for conciseness.

It is convenient in Haskell to use pattern matching to obtain the datum contained in a value of the type system, as is done for instance in the `square` and `swap` functions.

The function `C(n, r)` makes use of the factorial function from module `V1`. Note how the `rt` argument to `describeModule` function is passed to the function `c`. The function `c` first uses the `sym` function from the `RuntimeInterface` structure to obtain a representation of the `factorial` function. It then makes use of the `apply_proc` function to apply the procedure; the `apply_proc` function simplifies the code in two ways:

1. it takes the arguments in a list, so that the user need not consider the "folded" representation of functions, and

2. it evaluates the resulting IO on behalf of the caller.

To illustrate the second point, we note that the factorial function from module `V` is in fact represented as a procedure – that is, it is of type `Int -> IO Int` rather that just `Int -> Int`. (This is a peculiarity of the V language, which does not permit the definition of pure functions.) Thus, merely applying the factorial function returns a value of type `IO Int`, which must be applied separately to obtain a value. This application if preformed by `apply_proc`.

7.7 The V Module

If V is the least powerful of the four languages used in this chapter, it is, however, the most convenient for the current demonstration.

The syntax of the V language provides special constructs to facilitate programming in MLPE; in particular:

- a "module" construct, in which the name of the module is indicated and the exported symbols are listed,
- an "import" construct, to explicitly import modules,
- a "func" construct, to define procedures,
- a "dot-notation" to resolve symbols from imported modules.

In addition, a direct correspondence is established between the types and values in V and those of MLPE, so that no explicit calls or type casts are needed, as was the case with previous languages.

As can be seen from the source code (Appendix 8), the resulting code is much less encumbered than equivalent code in C, Java or Haskell.

The module `V1` defines a number of simple data elements and a number of functions. Every exported symbol is listed in the module header. Simple values are associated with symbols using explicit assignments in the top-level scope.

Functions definitions include the identification of the type of the arguments and return values. Functions return a value using the "return" construct.

The `V1` module provides versions of the `swap`, `hello` and `map` functions. Even though our simplistic modeling of the V language (Section 6.2) did not include a particular provision for higher-order functions, its lack of type checking allows the passing of functions as arguments. However the V language does not allow to specify a parameter of function type, so we simply let it `Void`. For instance, the type of the `map` function is set to `Void -> [Void] -> [Void]`.

In addition to the above three function, the `V1` modules defines

- the function `plusOne` adds one to its integer argument;
- the function `factorial` computes the factorial of an integer;
- the functions `timesFour` multiplies its integer argument by four;
- the function `doubleAll` doubles every integer component of a list;

The following shell session shows how the module presents itself in MLPE and how the above functions behave:

```
mlpe# :m V1
Module V1 [C1]
  doubleAll = {[Int] -> IO [Int]}
  factorial = {Int -> IO Int}
  hello = {IO Void}
  l = ["a", "b", "c", "d", "e"]
  map = {Void -> [Void] -> IO [Void]}
  n = 24
  p = (0, 10)
  plusOne = {Int -> IO Int}
  s = "less is more"
  swap = {(Void, Void) -> IO (Void, Void)}
  timesFour = {Int -> IO Int}
mlpe# V1.plusOne 24
  ==> 25
mlpe# V1.map V1.factorial [1,2,3,4,5]
  ==> [1, 2, 6, 24, 120]
mlpe# V1.map V1.timesFour [1,2,3,4,5]
  ==> [4, 8, 12, 16, 20]
mlpe# V1.doubleAll [1,2,3,4,5]
  ==> [2, 4, 6, 8, 10]
```

Chapter 8 Related Work

A full-scale attempt at supporting multi-language programming is found in the Microsoft .NET system. The system is founded on the capabilities of the Common Language Infrastructure (CLI) [MG], which incorporates a virtual machine meant to serve as a target for multiple source languages. It is a multi-threaded machine, with its own type system and intermediate language. Its type system is similar in scope to ours, but richer. (Like ours, it makes use of opaque references for the use of foreign runtime systems.)

Implementing full support for a language in .NET involves re-targeting its compiler to the CLI virtual machine. However, it is possible to allow a language to inter-operate with .NET without doing any change to the implementation, by simply writing a library for accessing .NET. The implementation task may then be similar to what we have done when implementing support for languages using black-box implementations. One example of this is Hugs98 for .NET [Finne03].

Inter-operability of object-oriented languages has been developed and used extensively. The Common Object Request Broker Architecture (CORBA) is a standard that provides connectivity for object-oriented languages. It defines an Interface Definition Language (IDL) for the specification of services in a language-independent manner. In addition to cross-language operability, CORBA allows execution in a distributed environment. The Inter-Language Unification System (ILU) [Janssen00] from Xerox is another multi-language interface system for object-oriented languages.

John Sturdy, in his Ph. D. thesis [Sturdy91], develops an architecture for language interpreters that allows for mixed-language programming. It goes beyond cross-language operability in that the language implementations themselves are built out of a common set of abstractions. It defines a common framework for interpretation based on a common representation of programs, program states and languages. (Programs can be uniformly represented as parse trees, whose nodes contain references to operators; languages themselves are modeled as mappings from operator names to meaning.) The system allows fully transparent inter-language calling; it also allows flexibility in semantic modeling, as a language can be extended by adding operators to it, usually

independently of other constructs. Sturdy's system provides reflective capabilities that MLPE does not.

Lambada [MF00] provides inter-operability between Haskell and Java. Like our support of Java, it is programmed directly through the JNI and the FFI. It is meant to make it more convenient to access Java objects from the Haskell side. For instance, it defines type classes for types that can be marshaled to and from Java, so that arguments to methods and return values can be converted automatically.

Our approach to cross-language operability bears some similarity to the model-based approach to user interface design, a current tendency of industry research. The implementation of a user interface can be generated automatically from a user interface model (typically expressed in XML) by a tool. Moreover, the implementations of the individual UI components can be generated in different languages, and these components can regardless inter-operate at runtime. This may be achieved by having the individual components operate over a set of common runtime abstractions that model UI concepts. In the same way, in MLPE, modules written in different languages inter-operate through a set of common runtime abstractions.

A large number of tools based on various formal methods for defining syntax and semantics have been developed in the past. Several of them are enumerated in [HK00].

We derived our syntactic processing facility from Happy [Happy], which is a parser generator for Haskell that is similar to "yacc". There also exits a scanner generator for Haskell similar to "lex" [Alex], but we didn't use it.

The fact that the specifications are meant to be expressed directly in the implementation language, without recourse to external tools, is uncommon in such tools. Note however that our prototyping tool has this feature in common with parsing combinator libraries (see e.g. [HM96].)

Chapter 9 Conclusion

MLPE is a programming environment that facilitates experiments with mixed-language programming and language prototyping. It provides a set of common abstractions – a common type system and module system – through which languages of diverse nature can be made to interoperate.

The chief advantage of MLPE is extensibility. By implementing support for a number of languages, we have shown that MLPE could be extended to support a particular language with moderate effort. We have shown that it can accommodate diverse implementation technologies, such as native code (C), a virtual machine (Java), and a prototype interpreter (V.)

MLPE is also highly integrated. It exposes a simple interface through which the user can initiate the compilation and loading of modules written in the supported languages.

The language prototyping facility of MLPE allows the user to conveniently write the specifications of the syntax of a prototype language as part of its implementation so that no external tool is needed. The treatment of syntax being based on conventional techniques (ordinary lexical specifications, LALR grammars), the prototyping tool is easy to use to language implementers.

MLPE is particularly suited for the use of specialized languages (or domain-specific languages) in the context of larger applications. The language prototyping tool makes it relatively easy to construct to the implementation of a simple language (such as V.) The implementation can then be integrated into MLPE with moderate effort. The language in question can then be used to access functionality implemented in any other language supported by MLPE, and *vice-versa*. Thus, much functionality is "inherited" in the newly defined language, at little cost.

Programming MLPE itself was an interesting experiment. The choice of a purely functional language was somewhat unconventional for a project that involved a significant amount of system programming. Haskell proved very suitable for the purpose of implementing the main abstractions of MLPE. It did, however, slightly complicate the interfacing of the Java runtime. We appreciated the greatest benefits of

Haskell when writing the prototype implementation of the V language, which was a straightforward translation of its (informally stated) semantics.

The project was indeed rather large in scope. We managed to design an integrated, extensible and interactive programming environment by assembling a relatively small number of key abstractions (the module system, type system, and language processors.) In counterpart, we had to compromise on some other aspects in the scope of this thesis.

Using MLPE to program using general-purpose languages (C, Java and Haskell) may appear tedious. The programmer has to write additional modules headers, write explicit calls (and type casts) for manipulating basic data, and do special calls for accessing foreign symbols. In our treatment of these languages, the focus has been on providing the languages with *semantics* for interacting with other languages in a generic manner, and the most direct way to do it is through a support library.

In contrast to general purpose languages, the specialized language that we have designed, V, was more tightly integrated with MLPE. Through a direct correspondence between the constructs of V and the runtime abstractions of MLPE, modules in V could be programmed in a much less encumbered style.

9.1 Future Work

The design of MLPE does not preclude a more convenient interfacing with general-purpose languages, so as to make the implementation of modules in these languages more practical. This could be achieved by using more advanced features of the object languages. For instance, in C, macros could be used to make the definition of functions and module headers more succinct. In Java, the refection API could be used to adapt Java functions to the common type system automatically, and to derive entire module definitions from class definitions. In Haskell, type classes could be used to marshal between Haskell types and the common type system automatically.

MLPE's type system is largely "primitive", as it was designed to accommodate various language programming paradigms. In particular, the suitability of MLPE's type system as a basis for inter-operability of object-oriented languages may be investigated. The inclusion of opaque types in the type system was meant as a provision for such application.

The user interface of MLPE could be greatly improved. For this purpose, one may consider the integration of MLPE into Eclipse [Eclipse]. Eclipse is an Integrated Development Environment (IDE) that can be used to develop applications using diverse languages. It provides additional extension points for additional editing modes and tools. Thus, it appears that Eclipse's extensibility is complementary to that of MLPE.

References

[Alex] *Alex: A lexical analyzer generator for Haskell.* Accessed on the 19th of July, 2004. URL http://www.haskell.org/alex/.

[BMW92] D.F. Brown, H.Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator. In *CC'92, Proceedings of the 4th International Conference on Compiler Construction, Paderborn*, Lecture Notes in Computer Science 641, pages 95—109, Sprigner-Verlag, 1992.

[Cardelli97] Luca Cardelli. Type Systems. In *The Computer Science and Engineering Handbook.* CRC Press, 1997.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to algorithms*, second ed. The MIT Press, 2001.

[CORBA] *The OMG's CORBA Website.* URL http://www.corba.org. Accessed on the 14th of September, 2004.

[Eclipse] *Eclipse.* URL http://www.eclipse.org. Accessed on the 15th of September, 2004.

[Espinosa95] Espinosa, D.A. *Semantic Lego.* Ph.D. Thesis, Columbia University, Department of Computer Science, 1995.

[FFI] Manuel Chakravarty (ed.) and others. *The Haskell 98 Foreign Function Interface 1.0.* Accessed on the 12th of August, 2004. URL http://www.cse.unsw.edu.au/~chak/haskell/ffi/ffi.pdf

[Finne03] Sigbjorn Finne. *Hugs98 for .Net.* Accessed on the 28th of July, 2004. URL http://galois.com/~sof/hugs98.net/

[Janssen00] Bill Janssen. *Inter-Language Unification – ILU.* Accessed on the 28th of July, 2004. URL ftp://parcftp.parc.xerox.com/pub/ilu/ilu.html

[Happy] *Happy: The parser generator for Haskell.* Accessed on the 19th of July, 2004. URL http://www.haskell.org/happy/.

[HK00] Jan Heering and Paul Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39-48, March 2000.

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. *Journal of functional programming*, 8(4):437-444, 1996.

[LB98] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications* (OOPSLA'98), pages 36-44, 1998.

[LHJ95] Sheng Liang, Paul Hudak and Mark Jones. *Monad transformers and modular interpreters.* In Conference record of POPL '95, 22nd {ACM} SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22—25. ACM Press, 1995

[Liang99] S. Liang. *The Java Native Interface.* Addison Wesley Longman. 1999.

[Linz01] Peter Linz. *An introduction to formal languages and automata, 3rd ed.* Jones and Barlett Publishers, 1990.

[Make] Free Software Foundation. *GNU Make*. URL http://www.gnu.org/software/make/

[MF00] Erik Meijer and Sigbjorn Finne. Lambada, Haskell as a Better Java, In *Proc. Haskell Workshop 2000.*

[MG] Erik Meijer and John Gough. *Technical Overview of the Common Language Runtime*. Accessed on the 28th of July, 2004. URL http://citeseer.lcs.mit.edu/meijer00technical.html

[Moggi90] Eugenio Moggi. *An abstract view of programming languages*. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.

[Mosses76] P.D. Mosses. Compiler generation using denotational semantics. In M. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 45, pages 436-441. Springer, Berlin, 1976.

[Mosses92] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mosses94] Peter D. Mosses. *A Tutorial on Action Semantics - Notes for FME'94*. Accessed on the 19th of July, 2004. URL http://citeseer.ist.psu.edu/mosses94tutorial.html.

[NN88] F. Nielson and H. Riis Nielson. Two-level semantics and code generation. Theoretical Computer Science, 56(1):59-133, January 1988.

[Ørbæk94] Peter Ørbæk. OASIS: An optimizing action-based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *Lecture Notes in Computer Science*, pages 1-15. Springer-Verlag, 1994.

[Parsons92] Thomas W. Parsons. *Introduction to Compiler Construction*. W.H. Freeman and Company, 1992.

[Ram] Hampus Ram. *Dynamic Linking in Haskell*. Accessed on the 28th of July, 2004. URL http://www.dtek.chalmers.se/~d00ram/dynamic/dynamiclinker.pdf

[Schmidt95] David A. Schmidt. *Programming Language Semantics*. ACM Computing Surveys. 1995.

[Schmidt97] D. A. Schmidt, *On the need for a popular formal semantics*, ACM SIGPLAN Notices 32 (1) (1997) 115-116.

[Stoy77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977

[Sturdy91] John Sturdy. *A Lisp through the Looking Glass*. Accessed on the 29th of July, 2004. URL http://www.cb1.com/~john/thesis/thesis.html

Appendix 1. Syntax of the Shell Language

```
shell_lexicon = Lexicon
                [LRS "NAME"       identifiers,
                 LRS "NUMBER"     (re_Plus digit),
                 LRS "."          (RE_Set (st_single '.')),
                 LRS ","          (RE_Set (st_single ',')),
                 LRS "("          (RE_Set (st_single '(')),
                 LRS ")"          (RE_Set (st_single ')')),
                 LRS "["          (RE_Set (st_single '[')),
                 LRS "]"          (RE_Set (st_single ']')),
                 LRS "$"          (RE_Set (st_single '$')),
                 LRS "left_arrow" (re_String "<-"),
                 LRS "STRING" strings]
              (RE_Star whitespace)  -- ignored input


shell_concrete_grammar_spec =
  ConcreteGrammarSpec
    -- target (abstract) grammar
    no_abstract_grammar_spec
    -- terminal symbols
    (get_terminals shell_lexicon)
    -- non-terminal symbols
    [NTS "Start"  "Term"
         [RLS "NAME<1> left_arrow Exp+<2>"      "binding",
          RLS "Exp+<>"                          ""],
     NTS "Exp+"  "Term"
         [RLS "Exp<>"                           "",
          RLS "Exp+<1> Exp<2>"                  "app"],
```

```
NTS "Exp"  "Term"
    [RLS "QualifiedName<1>"                "qualified_name",
     RLS "NUMBER<1>"                       "number",
     RLS "STRING<1>"                       "string",
     RLS "( Exp+<> )"                      "" ,
     RLS "( Exp+<1> , Exp+<2> )"           "pair",
     RLS "[ ]"                             "emptyList",
     RLS "[ Lst<> ]"                       ""],
NTS "Lst"     "Lst"
     [RLS "Exp+<1> , Lst<2>"            "list",
      RLS "Exp+<1> Null<2>"                "list"],
NTS "Null"    "Lst"
    [RLS ""                               "emptyList"],
NTS "QualifiedName"  "QualifiedName"
    [RLS "NAME<1> . QualifiedName<2>"      "extension",
     RLS "NAME<1>"                         "name"]]
```

Appendix 2. Implementation of the V Language

```
module V (
  V(..), T(..), semC, v_parse,
  V_Inst, v_builtin_procedures, v_init, v_state, v_update, v_apply_proc
) where

import Data.IORef
import CSet
import CMap
import RegExp
import Lex
import AbstractGrammar
import ConcreteGrammar
import PostParse
import Types

-- lexical specifications
v_lexicon =
  Lexicon
     [LRS "+"      (re_String "+"),
      LRS "-"      (re_String "-"),
      LRS "*"      (re_String "*"),
      LRS "="      (re_String "="),
      LRS ";"      (re_String ";"),
      LRS ":"      (re_String ":"),
      LRS ","      (re_String ","),
      LRS "."      (re_String "."),
      LRS "{"      (re_String "{"),
```

```
      LRS "}"      (re_String "}"),
      LRS "("      (re_String "("),
      LRS ")"      (re_String ")"),
      LRS "["      (re_String "["),
      LRS "]"      (re_String "]"),
      LRS ":="     (re_String ":="),
      LRS "if"     (re_String "if"),
      LRS "then"   (re_String "then"),
      LRS "else"   (re_String "else"),
      LRS "func"   (re_String "func"),
      LRS "return" (re_String "return"),
      LRS "module" (re_String "module"),
      LRS "where"  (re_String "where"),
      LRS "import" (re_String "import"),
      LRS "Int"    (re_String "Int"),
      LRS "String" (re_String "String"),
      LRS "Void"   (re_String "Void"),
      LRS "INT"    (re_Plus digit),
      LRS "VAR"    id_pattern,
      LRS "STRING" strings]
    (re_Plus $ RE_Union (RE_Star whitespace)     -- ignored input
                        comment)
comment =
   (re_String "--") `RE_Cat` (re_Plus non_newline)
 where non_newline = RE_Set $ (st_set ['\0'..'\255']) `st_diff` (st_set
['\n'])


id_pattern =
```

```
   (RE_Cat (RE_Set (st_set (['a'..'z'] ++ ['A'..'Z'] ++ "_'")))
           (RE_Star $ RE_Set (st_set (['a'..'z'] ++ ['A'..'Z'] ++
['0'..'9'] ++ "_'"))))


strings :: RegExp Char
strings =
  (RE_Cat (RE_Set (st_single '\"'))
          (RE_Cat    (RE_Star    (RE_Set    (st_set    (['\0'..'!']    ++
['#'..'\255'])))))
                  (RE_Set (st_single '\"'))))


-- concrete syntax
v_concrete_grammar_spec =
  ConcreteGrammarSpec
    -- target (abstract) grammar (unspecified)
    no_abstract_grammar_spec
    -- terminal symbols
    (get_terminals v_lexicon)
    -- non-terminal symbols
    [NTS "Cmd"    "Cmd"
        [RLS "Cmd<1> ; Cmd'<2>"                              ";",
         RLS "Cmd'<>"                                        ""],
     NTS "Cmd'"   "Cmd"
        [RLS "VAR<1> := Exp<2>"                              ":=",
         RLS "module VAR<1> ( Elist<2> ) where Cmd'<3>"  "module",
         RLS "if Exp<1> then Cmd<2> else Cmd<3>" "if",
         RLS "import VAR<1>"                    "import",
         RLS "return Exp<1>"                    "return",
         RLS "func VAR<1> ( Plist<2> ) : Type<3> { Cmd<4> }"  "func",
```

```
    RLS "Exp<1>"                          "exp"],
NTS "Exp"   "Exp"
   [RLS "Exp'<1> = Exp'<2>"               "=",
    RLS "Exp'<>"                          ""],
NTS "Exp'"  "Exp"
   [RLS "Exp'<1> + Exp''<2>"               "+",
    RLS "Exp'<1> - Exp''<2>"               "-",
    RLS "Exp''<>"                          ""],
NTS "Exp''"    "Exp"
   [RLS "Exp''<1> * Exp'''<2>"             "*",
    RLS "Exp'''<>"                         ""],
NTS "Exp'''"   "Exp"
   [RLS "INT<1>"                          "n",
    RLS "STRING<1>"                       "s",
    RLS "VAR<1>"                          "x",
    RLS "( Exp<1> , Exp<2> )"             "pair",
    RLS "( Exp<1> : Exp<2> )"             "cons",
    RLS "[ ]"                             "emptyList",
    RLS "[ Lst<> ]"                       "",
    RLS "( Exp<> )"                       "",
    RLS "Exp'''<1> ( Alist<2> )"           "call",
    RLS "VAR<1> . VAR<2>"                 "."],
NTS "Lst"     "Exp"
   [RLS "Exp<1> , Lst<2>"                 "list",
    RLS "Exp<1> Null<2>"                  "list"],
NTS "Null"   "Exp"
   [RLS ""                                "emptyList"],
NTS "Type"   "Type"
```

```
   [RLS "Int"                              "Int",
    RLS "String"                           "String",
    RLS "( Type<1> , Type<2> )"            "Pair",
    RLS "[ Type<1> ]"                      "List",
    RLS "Void"                             "Void" ],
-- parameter list
NTS "Plist"  ""
   [RLS "Plist<1> , Plist'<2>"             "P,",
    RLS "Plist'<>"                         ""],
NTS "Plist'" ""
   [RLS "VAR<1> : Type<2>"                 "P",
    RLS ""                                 "noP"],
-- argument list
NTS "Alist"  ""
   [RLS "Alist<1> , Alist'<2>"             "A,",
    RLS "Alist'<>"                         ""],
NTS "Alist'" ""
   [RLS "Exp<1>"                           "A",
    RLS ""                                 "noA"],
-- exported symbol list
NTS "Elist"  ""
   [RLS "Elist<1> , Elist'<2>"             "E,",
    RLS "Elist'<>"                         ""],
NTS "Elist'" ""
   [RLS "VAR<1>"                           "E",
    RLS ""                                 "noE"]]
```

```
-- the V parser
v_parse =
  generateParser v_lexicon
                 no_abstract_grammar_spec -- unspecified
                 v_concrete_grammar_spec


-- Semantic domains
-- values
data V  = V'N Int                                  -- integer
        | V'S String                               -- string
        | V'P V V
        | V'L [V]                                  -- non-empty list
        | V'F T (State -> [V] -> IO (V, State))    -- procedure
        | V'V                                      -- void
        | V'U                                      -- undefined
-- types
data T  = T'Int
        | T'String
        | T'Pair T T
        | T'List T
        | T'Func [T] T
        | T'Void
        | T'U              -- undefined / ill-types
type State = Map_ String V
```

```
-- syntactic categories

type Exp = Term

type Cmd = Term


-- semantic functions

semV :: Exp -> State -> IO (V, State)

semV (Term "n" [Atom "INT" n])    = \ state -> return $ (V'N (read n),
state)

semV (Term "s" [Atom "STRING" s])= \ state ->

       return $ (V'S ((reverse . tail . reverse . tail) s), state)

semV (Term "x" [Atom "VAR" x])    = \ state ->

      case (mp_lookup state x) of

         Just v  -> return (v, state)

         Nothing -> return (V'U, state)

semV (Term "emptyList" [])        = \ state -> return (V'L [], state)

semV (Term "list" [x, y])         = \ state ->

        do (head, state')      <- semV x state

           (V'L tail, state'') <- semV y state'

           return ((V'L (head:tail)), state'')

semV (Term "+" [x, y])    = \ state ->

        do (x_v, state')  <- semV x state

           (y_v, state'') <- semV y state'

           case (x_v, y_v) of

             (V'N x_n, V'N y_n) -> return (V'N (x_n + y_n), state'')

             _         -> do putStrLn "non-numeric argument to +"

                             return (V'U, state'')

semV (Term "-" [x, y])    = \ state ->

        do (x_v, state')  <- semV x state
```

```
            (y_v, state'') <- semV y state'
            case (x_v, y_v) of
             (V'N x_n, V'N y_n) -> return (V'N (x_n - y_n), state'')
             _                  -> do putStrLn "non-numeric argument to -"
                                      return (V'U, state'')
semV (Term "*" [x, y])   = \ state ->
        do (x_v, state')  <- semV x state
           (y_v, state'') <- semV y state'
           case (x_v, y_v) of
             (V'N x_n, V'N y_n) -> return (V'N (x_n * y_n), state'')
              _            -> do putStrLn "non-numeric argument to *"
                                 return (V'U, state'')
semV (Term "=" [x, y])   = \ state ->
        do (x_v, state')  <- semV x state
           (y_v, state'') <- semV y state'
           if x_v == y_v
              then return (V'N 1, state'')
              else return (V'N 0, state'')
semV (Term "pair" [x, y])        = \ state ->
        do (x_v, state')  <- semV x state
           (y_v, state'') <- semV y state'
           return ((V'P x_v y_v), state'')
semV (Term "cons" [h, t])        = \ state ->
        do (h_v,     state')  <- semV h state
           (V'L t_v, state'') <- semV t state'
           return (V'L (h_v:t_v), state'')
semV (Term "call" [func, args']) = \ state ->
        do  (func', state') <- semV func state
```

```
            case func' of
              (V'F _ p) ->
                 do let args      = arg_list args'
                    (args_v, state'') <- eval_args (args, state)
                    (ret, state''') <- p state' args_v
                    return (ret, state''')
              _ -> do putStrLn ("is not a procedure: " ++ show func)
                      return (V'U, state)
semV (Term "." [Atom "VAR" mod, Atom "VAR" sym]) = \ state ->
       do case mp_lookup state "import_context" of
            Just (V'F _ f) -> do (ret, state') <-
                                        f state [V'S mod, V'S sym]
                                return (ret, state')
            Nothing   -> do putStrLn "can't find import context!"
                            return (V'U, state)

semC :: Cmd -> State -> IO State
semC (Term "exp" [exp]) = \ state -> do (_, state') <- semV exp state
                                       return state'
semC (Term "write" [e])              = \ state ->
       do (e_v::V, state') <- semV e state
          return state'
semC (Term ":="   [Atom "VAR" x, e]) = \ state ->
       do (e_v::V, state') <- semV e state
          return $ mp_update state' x e_v
semC (Term "module" [Atom "VAR" x, exports, cmd]) = \ state ->
       do let state'  = mp_update state "module_name" (V'S x)
              state'' = mp_update state' "exported_symbols"
```

```
                                        (V'L (export_list exports))
          state'''   <- semC cmd state''
          return state'''
semC (Term "import" [Atom "VAR" x]) = \ state ->
        do let imports = mp_lookup state "imports"
               imports' = case imports of
                            Nothing      -> V'L [V'S x]
                            Just (V'L i) -> V'L (V'S x:i)
          return $ mp_update state "imports" imports'
semC (Term "return" [ret]) = \ state ->
        do (ret_v, state') <- semV ret state
          return $ mp_update state' "ret" ret_v
semC (Term "func" [Atom "VAR" name, params', type_, cmd]) = \ state ->
  do let params = param_list params'
         f = \ state' -> \ args ->
             do let bindings = zip params args
                    state''  = foldl ( \state -> \(param, arg) ->
                                         mp_update state param arg )
                                     state'
                                     bindings
                state'''     <- semC cmd state''
                let ret      = case mp_lookup state''' "ret" of
                                 Just x  -> x
                                 Nothing -> V'U
                return (ret, state''')
        f_type = T'Func (param_type_list params') (readType type_)
     return (mp_update state name (V'F f_type f))
semC (Term ";" [c1, c2])             = \ state ->
```

```
        do state' <- semC c1 state
           semC c2 state'
semC (Term "if" [c, t, e]) = \ state ->
    do (c', state') <- semV c state
       case c' of
         V'N 0  -> do semC e state'
         _      -> do semC t state'
semC x = \ state ->
        do putStrLn $ "unhandled command: " ++ (show x)
           return state


-- built-in V procedures
left  = V'F
          (T'Func [T'Pair T'Void T'Void] T'Void)
          (\ state -> \ [V'P l r] -> return (l, state))
right = V'F
          (T'Func [T'Pair T'Void T'Void] T'Void)
          (\ state -> \ [V'P l r] -> return (r, state))
head_ = V'F
          (T'Func [T'List T'Void] T'Void)
          (\ state -> \ [V'L (h:t)] -> return (h, state))
tail_ = V'F
          (T'Func [T'Pair T'Void T'Void] T'Void)
          (\ state -> \ [V'L (h:t)] -> return ((V'L t), state))
print = V'F
          (T'Func [T'Pair T'Void T'Void] T'Void)
          (\ state -> \ [v] ->
              do case v of
```

```
                    V'S s -> putStr s
                    x     -> putStr (show x)
                return (V'V, state))
printLn = V'F
          (T'Func [T'Pair T'Void T'Void] T'Void)
          (\ state -> \ [v] ->
            do case v of
                    V'S s -> putStrLn s
                    x     -> putStrLn (show x)
               return (V'V, state))
v_builtin_procedures :: State =
   mp_map [("left",    left),
           ("right",   right),
           ("head",    head_),
           ("tail",    tail_),
           ("print",   V.print),
           ("printLn", V.printLn)]

-- high-level wrapper around the semantic functions/interpreter
type V_Inst = IORef State

v_init :: String -> State -> IO V_Inst
v_init program state =
  do state' <- semC (v_parse program) state
     inst   <- newIORef state'
     return inst

v_state :: V_Inst -> IO State
```

```
v_state inst = readIORef inst


v_update :: V_Inst -> State -> IO ()
v_update inst state = writeIORef inst state


v_apply_proc :: V_Inst -> V -> [V] -> IO V
v_apply_proc i (V'F _ f) args =
  do state    <- v_state i
     (v', s') <- f state args
     v_update i s'
     return v'


-- helper functions / instance defitions


instance Eq V where
    (V'N x) == (V'N y)      = x == y
    (V'S x) == (V'S y)      = x == y
    (V'P a b) == (V'P c d) = (a, b) == (c, d)
    (V'L x) == (V'L y)      = x == y
    V'V     == V'V          = True
    _       == _            = False


instance Show V where
    show (V'N n) = show n
    show (V'S s) = "\"" ++ s ++ "\""
    show (V'P x y) = "(" ++ show x ++ ", " ++ show y ++ ")"
    show (V'L elems) = "[" ++ comma (map show elems) ++ "]"
    show (V'F t _) = "<proc:(" ++ show t ++ ">"
```

```
    show V'U      = "<undefined>"


comma :: [String] -> String
comma [] = ""
comma [a] = a
comma (h:t) = h ++ ", " ++ (comma t)


instance Show T where
    show T'Int         = "Int"
    show T'String      = "String"
    show (T'Pair s t) = "(" ++ show s ++ ", " ++ show t ++ ")"
    show (T'List s)    = "[" ++ show s ++ "]"
    show (T'Void)      = "Void"
    show (T'Func ps rt) = "(" ++ (foldl (++) ""
                                        (separate (map show ps) ", ") )
                              ++ ")->" ++ show rt ++  ">"
    show T'U           = "<indefined type>"


separate :: forall a. [a] -> a -> [a]
separate (h:(h':t)) s = (h:(s:(separate (h':t) s)))
separate l _ = l


param_list :: Term -> [String]
param_list (Term "P," [s, t]) = (param_list s) ++ (param_list t)
param_list (Term "P" [Atom "VAR" x, _]) = [x]
param_list (Term "noP" _) = []


param_type_list :: Term -> [T]
```

```
param_type_list (Term "P," [s, t]) = (param_type_list s)
                                   ++ (param_type_list t)
param_type_list (Term "P" [_, t])  = [readType t]
param_type_list (Term "noP" _)  = []

arg_list :: Term -> [Term]
arg_list (Term "A," [s, t]) = (arg_list s) ++ (arg_list t)
arg_list (Term "A"  [x])    = [x]
arg_list (Term "noA" _)     = []

export_list :: Term -> [V]
export_list (Term "E," [s, t]) = (export_list s) ++ (export_list t)
export_list (Term "E" [Atom "VAR" x]) = [V'S x]
export_list (Term "noE" _) = []

readType :: Term -> T
readType (Term "Int" [])      = T'Int
readType (Term "String" [])   = T'String
readType (Term "Pair" [s, t]) = T'Pair (readType s) (readType t)
readType (Term "List" [s])    = T'List (readType s)
readType (Term "Void" [])     = T'Void

eval_args :: ([Exp], State) -> IO ([V], State)
eval_args ([], state) = return ([], state)
eval_args ((arg0:args), state) =
  do (arg0_v, state')  <- semV arg0 state
     (args_v, state'') <- eval_args (args, state')
     return ((arg0_v:args_v), state'')
```

Appendix 3. Implementation of the V Module Loader

```
module VMod (
   v_module_loader
) where
import Data.IORef
import CSet
import CMap
import Util
import Project
import RT
import PE
import V

-- marshaling functions

marshalType :: CommonType -> T
marshalType CT_Int        = T'Int
marshalType CT_String     = T'String
marshalType (CT_Pair s t) = T'Pair (marshalType s) (marshalType t)
marshalType (CT_List s)   = T'List (marshalType s)
marshalType f_t@(CT_Func s t) =
  T'Func (map marshalType params) (marshalType ret')
  where list   = func_type_list f_t
        params = (reverse.tail.reverse) list
        ret    = last list
        ret'   = case ret of
                   (CT_IO t) -> t
                   t         -> t
```

```
        func_type_list (CT_Func s t) = s:(func_type_list t)
        func_type_list t             = [t]
marshalType (CT_IO t)     = T'Func [] (marshalType t)
marshalType CT_Void       = T'Int

unmarshalType :: T -> CommonType
unmarshalType T'Int        = CT_Int
unmarshalType T'String     = CT_String
unmarshalType (T'Pair s t) = CT_Pair (unmarshalType s)
                                     (unmarshalType t)
unmarshalType (T'List s)   = CT_List (unmarshalType s)
unmarshalType (T'Func p r) = foldr CT_Func
                                   (CT_IO (unmarshalType r))
                                   (map unmarshalType p)
unmarshalType (T'Void)     = CT_Void
unmarshalType (T'U)        = CT_Void

marshalObject :: V_Inst -> CommonObject -> V
marshalObject _ (CO_Int i)    = V'N i
marshalObject _ (CO_String s) = V'S s
marshalObject i (CO_Pair a b) = V'P (marshalObject i a)
                                    (marshalObject i b)
marshalObject i (CO_EmptyList)= V'L []
marshalObject i l@(CO_List _ _) = V'L (map (marshalObject i)
                                           (readCO_List l))
marshalObject i (CO_IO t io) =
     V'F (T'Func [] (marshalType t))
         (\state -> \ args_v ->
```

```
                do ret  <- io
                   let ret' =  marshalObject i ret
                   return (ret', state))
marshalObject i o@(CO_Func s t f)
   = V'F (marshalType (CT_Func s t))
         (\ state -> \ args_v ->
             do let args = map (unmarshalObject i) args_v
                ret  <- apply o args
                ret' <- case ret of
                          CO_IO _ io -> io
                          x          -> return x
                let ret'_v = marshalObject i ret'
                return (ret'_v, state))
marshalObject _ x = V'S ("unhandled in marshalObject: " ++ show x)

unmarshalObject :: V_Inst -> V -> CommonObject
unmarshalObject _ (V'N i)     = CO_Int i
unmarshalObject _ (V'S s)     = CO_String s
unmarshalObject i (V'P a b)   = CO_Pair (unmarshalObject i a)
                                        (unmarshalObject i b)
unmarshalObject i (V'L elems) = makeCO_List (map (unmarshalObject i)
                                                 elems)
unmarshalObject i p@(V'F t f) = defer i (unmarshalType t) p []
unmarshalObject _ (V'V)       = CO_Void
unmarshalObject _ (V'U)       = CO_Void
unmarshalObject _ x = CO_String  ("unhandled in unmarshalObject: "
                                        ++ show x)
```

```
-- the "defer" function - defers the application of a
-- multiple-argument function until every argument is supplied
defer :: V_Inst -> CommonType -> V -> [CommonObject] -> CommonObject
defer i (CT_Func s t) p parms =
     CO_Func s t
             (\p0  -> do return (defer i t p
                                      (parms ++ [p0])))
defer i (CT_IO t) p parms     =
     CO_IO t (do let parms_v = map (marshalObject i) parms
                 v_v     <- v_apply_proc i p parms_v
                 let v = unmarshalObject i v_v
                 return v)
defer i t p parms     =
     CO_IO t (do let parms_v = map (marshalObject i) parms
                 v_v     <- v_apply_proc i p parms_v
                 let v = unmarshalObject i v_v
                 return v)

-- the V loader
v_module_loader =
    Loader "V"
           -- loader:
         (\ dir -> \ (ModuleSpec mod_name source_file) ->
           do -- read the source file
              program_text <- readFile source_file
              -- instantiate the interpreter
              inst  <- v_init program_text v_builtin_procedures
```

```
-- create a surrogate import context
-- in the form of a V procedure
(import_context_ref::IORef ImportContext)
    <- newIORef (\ _ -> \ _ -> return CO_Void)
let import_context_proc =
        V'F (T'Func [T'String, T'String] T'Void)
            (\ state -> \ [V'S mod, V'S sym] ->
                do ctx <- readIORef import_context_ref
                   obj <- ctx mod sym
                   let obj_v = marshalObject inst obj
                   return (obj_v, state))
-- add the import context to the interpreter state
state <- v_state inst
v_update inst (mp_update state
                         "import_context"
                         import_context_proc)
-- obtain the list of imported modules and exported
-- values from the interpreter state
let list   (V'L l) = l
    string (V'S s) = s
    imports
       = map string (list (mp_get state "imports"))
    exported_symbols
       = map string
             (list (mp_get state "exported_symbols"))
    values
       = map (\ sym ->
                  case mp_lookup state sym of
```

```
                    Just v  -> (sym,
                                unmarshalObject inst v)
                    Nothing -> (sym, CO_Void))
            exported_symbols
-- assemble the module's internal representation
let mod  = Module mod_name imports
                (st_empty) (mp_map values)
return (-- internal representation of the module
        mod,
        -- import context handler
        (\ ctx -> writeIORef import_context_ref ctx)))
```

Appendix 4. Sample Project Specification

```
module ProjectSpec (
  spec
) where
import Project


spec =
  ProjectSpec
    "P1"  -- project name
    ""    -- home directory (unspecified, use current directory.)
    "C1"  -- default target module
    -- source files  name          language
    [SourceFileSpec  "C1.c"       "C",
     SourceFileSpec  "J1.java"    "Java",
     SourceFileSpec  "H1.hs"      "Haskell",
     SourceFileSpec  "V1.v"       "V"]
    -- resources    name        resource type         constituent  deps
    [ResourceSpec   "C1.o"     "C object file"       ["C1.c"]     [],
     ResourceSpec   "C1.so"    "shared C library"    ["C1.o"]     [],
     ResourceSpec   "J1.class" "Java class file"     ["J1.java"]  [],
     ResourceSpec   "H1.o"     "Haskell object file" ["H1.hs"]    []]
     -- modules      name    constituent
    [ModuleSpec     "C1"    "C1.so",
     ModuleSpec     "J1"    "J1.class",
     ModuleSpec     "H1"    "H1.o",
     ModuleSpec     "V1"    "V1.v" ]
```

Appendix 5. Sample C Module

```
#include <mlpe_runtime.h>

RT rt;   /* this module's import context */


/* some functions in the module */


/* double :: Int -> Int
   double x = 2 * x
 */
CommonObject *double_(CommonObject *o) {
   return coInt(2*coGetInt(o));
}


/* swap :: (S, T) -> (T, S)
   swap (a, b) = (b, a)
 */
CommonObject *swap(CommonObject *p) {
  return coPair(coGetRight(p), coGetLeft(p));
}


/* pwd :: IO String
   returns the current process working directory
 */
CommonObject *pwd() {
  return coString(getenv("PWD"));
}


/* map :: (a -> b) -> [a] -> [b]
```

```
   applies a function to every component of a list
 */
CommonObject *map(CommonObject *f, CommonObject *l) {
  CommonObject* (*func)(CommonObject*) =
    (CommonObject* (*)(CommonObject*)) coGetFunc(f);

  if (coIsEmpty(l)) return coEmptyList();
   else return coList(func(coGetHead(l)),
                      map(f, coGetTail(l)));
}

/* hello :: IO ()
   displays the canonical "hello, world!"
 */
CommonObject *hello() {
  printf("Hello, world!\n");
  return coVoid();
}

/* a Counter "class"
   wraps a counter with a state and methods to increment,
   reset, and lookup the value of the counter
 */

typedef int* Counter;  /* the state is just an intger value */

/* newCounter :: IO Counter */
CommonObject *newCounter() {
```

```
  Counter c = (Counter) malloc(sizeof(int));
  *c = 0;
  return coObj(ctOpaqueType("Counter"), c);
}

/* increment :: Counter -> IO () */
CommonObject *increment(CommonObject *o) {
  Counter c = (Counter) coGetObj(o);
  *c += 1;
  return coVoid();
}

/* reset :: Counter -> IO () */
CommonObject *reset(CommonObject *o) {
  Counter c = (Counter) coGetObj(o);
  *c = 0;
  return coVoid();
}

/* getValue :: Counter -> IO Int */
CommonObject *getValue(CommonObject *o) {
  Counter c = (Counter) coGetObj(o);
  return coInt(*c);
}

/* timesNine :: Int -> Int
   multiplies its argument by nine by applying the triple()
   function from module J1 twice */
```

```
CommonObject *timesNine(CommonObject *i) {
  /* obtain the function value from module J1 */
  CommonObject* triple_value = rt("J1", "triple");

  /* obtain the C-function from the function value
     (we need to cast to a C-function of the actual arity) */
  CommonObject* (*triple)(CommonObject*) =
    (CommonObject* (*)(CommonObject*)) coGetFunc(triple_value);

  return triple(triple(i));
}

/* module header */
ModuleSpec describeModule() {
   return moduleSpec(
             "C1",                    /* module name */
             imports("J1", NULL),     /* imported modules */
             types("Counter", NULL),  /* exported opaque types */
                                      /* exported symbols follow */

             /* some values of primitive types */
             entry("n",      coInt(1000)),
             entry("s",      coString("<some C-string>")),

             /* a pair and a list */
             entry("p",      coPair(coString("Left"),
                                    coString("Right"))),
             entry("l",      coList(coString("a"),
```

```
                        coList(coString("b"),
                          coList(coString("c"),
                          coEmptyList()))))),
/* functions follow */

/* double :: Int -> Int */
entry("double",  coFunc(ctInt(), ctInt(), double_)),
/* swap :: (a, b) -> (b, a) */
entry("swap",  coFunc(ctPair(ctVoid(), ctVoid()),
                      ctPair(ctVoid(), ctVoid()),
                      swap)),
/* gewPWD :: IO String*/
entry("getPWD", coIO(ctString(), pwd)),
/* hello :: IO Void */
entry("hello", coIO(ctVoid(), hello)),

/* map :: (a -> b) -> [a] -> [b] */
entry("map", coFunc( ctFunc(ctVoid(), ctVoid()),
                     ctFunc(ctList(ctVoid()),
                            ctList(ctVoid())),
                     map)),

/* timesNine::Int -> Int
   (uses a function from module J1) */
entry("timesNine", coFunc(ctInt(), ctInt(), timesNine)),

/* functions that wrap "methods" of the Counter "class" */
entry("newCounter", coIO(ctOpaqueType("Counter"),
```

```
                                newCounter)),
            entry("increment", coFunc(ctOpaqueType("Counter"),
                                ctIO(ctVoid()),
                                increment)),
            entry("reset", coFunc(ctOpaqueType("Counter"),
                            ctIO(ctVoid()),
                            reset)),
            entry("getValue", coFunc(ctOpaqueType("Counter"),
                                ctIO(ctInt()),
                                getValue)),
            NULL);
}
```

Appendix 6. Sample Java Module

```
import rt.*;


public class J1 extends Module {
  /* module header */
    public CommonObject describeModule() {
      return spec(
        "J",                        /* module name              */
        new String[] {"H1"},        /* imported modules         */
        new String[] {"Toggle"},    /* exported opaque types    */
                                    /* exported symbols follow */
        new CommonObject[] {
          /* some primitive data */
        entry("n", new CO_Int(45)),
          entry("s", new CO_String("<a string in Java>")),


          /* a pair */
          entry("p", new CO_Pair(new CO_String("Red"),
                        new CO_String("Blue"))),
          /* a list */
          entry("l", new CO_List(new CO_Int(1),
                       new CO_List(new CO_Int(2),
                         new CO_List(new CO_Int(3),
                           new CO_List(new CO_Int(4),
                           new CO_EmptyList()))))),


          /* some functions */
          entry("triple", triple),
```

```
        entry("swap", swap),
        entry("getClassPath", getClassPath),
        entry("hello", hello),
        entry("map", map),
        entry("fourthPower", fourthPower),

        /* functions that wrap methods of the Toggle class */
        entry("newToggle", newToggle),
        entry("toggle", toggle),
        entry("getState", getState)
    });
}

// triple :: Int -> Int
CommonObject triple =
    new CO_Func(new CT_Int(), new CT_Int())
        {
           public CommonObject func(CommonObject o) {
              int i = ((CO_Int)o).getIntValue();
              return new CO_Int(3*i);
           }
        };

// swap :: (a, b) -> (b, a)
CommonObject swap =
    new CO_Func(new CT_Pair(new CT_Void(), new CT_Void()),
                new CT_Pair(new CT_Void(), new CT_Void()))
        {
```

```
            public CommonObject func(CommonObject o) {
                return new CO_Pair( ((CO_Pair)o).getRight(),
                                    ((CO_Pair)o).getLeft());
            }
        };

// getClassPath :: IO String
CommonObject getClassPath =
    new CO_IO(new CT_String())
        {
           public CommonObject func() {
             try {
                return(new CO_String(
                   System.getProperty("java.class.path",".")));
             }
             catch (RuntimeException e) {
                return new CO_String(
                   "<wasn't able to obtain class path>");
             }
           }
        };

// hello :: IO ()
CommonObject hello =
    new CO_IO(new CT_String())
        {
          public CommonObject func() {
             System.out.println("Hello, world!");
```

```
            return new CO_Void();
        }
    };

// map :: (a -> b) -> [a] -> [b]
CommonObject map =
  new CO_Func(new CT_Func(new CT_Void(), new CT_Void()),
              new CT_Func(new CT_List(new CT_Void()),
                          new CT_List(new CT_Void())))
  {
      public CommonObject func(CommonObject f, CommonObject l) {
         CO_Func function = (CO_Func) f;
         CO_List list     = (CO_List) l;

         if (l instanceof CO_EmptyList) return new CO_EmptyList();
         else return new CO_List(function.func(list.getHead()),
                                 func(f, list.getTail()));
      }
  };

// fourthPower :: Int -> Int
CommonObject fourthPower =
    new CO_Func(new CT_Int(), new CT_Int())
        {
           public CommonObject func(CommonObject n) {
              // obtain the function value from module H1
              CO_Func square = (CO_Func) rt.sym("H1", "square");
              return square.func(square.func(n));
```

```
        }
    };

CommonType toggle_type = new CT_OpaqueType("Toggle");

// newToggle :: IO Toggle
CommonObject newToggle =
    new CO_IO(new CT_OpaqueType("Toggle"))
        {
           public CommonObject func() {
              return new CO_Obj(toggle_type, new Toggle());
           }
        };

// toggle :: Toggle -> IO ()
CommonObject toggle =
    new CO_Func(toggle_type, new CT_IO(new CT_Void()))
        {
           public CommonObject func(CommonObject o) {
              Toggle t = (Toggle) ((CO_Obj) o).getObject();
              t.toggle();
              return new CO_Void();
           }
    };

// getState :: Toggle -> IO Int
CommonObject getState =
    new CO_Func(toggle_type, new CT_IO(new CT_Void()))
```

```
            {
                public CommonObject func(CommonObject o) {
                  Toggle t = (Toggle) ((CO_Obj) o).getObject();
                  return new CO_Int(t.getState());
                }
            };

  /*  a simple toggle class */
  class Toggle {
    int v;
    Toggle() {
      v = 0;
    }
    void toggle() {
      v = 1 - v;
    }
    int getState() {
      return v;
    }
  }
};
```

Appendix 7. Sample Haskell Module

```
module H1 (
  describeModule
) where
import HS_Runtime
import Time


describeModule :: RuntimeInterface -> ModuleDescriptor
describeModule rt =
    ModuleDescriptor
       "H1"                                    -- module name
        ["V1"]                                 -- imported modules
        []                                     -- types
                                               -- exported symbols
        [-- some primitive data
         ("n", CO_Int 23),
         ("s", CO_String "It's better with the full power a Haskell."),
         -- a pair and a list
         ("p", CO_Pair (CO_String "flold") (CO_String "foldr")),
         ("l", CO_List (CO_String "x") $
                 CO_List (CO_String "y") $
                   CO_List (CO_String "z") CO_EmptyList),
         -- some simple functions
         ("square", square),
         ("swap", swap),
         ("getYear", getYear),
         ("hello", hello),
         ("map", map_),
```

```
        ("C", c rt)]


-- square :: Int -> Int
square =
   foldFunc [CT_Int, CT_Int] $
     \ [CO_Int x] -> return $ CO_Int (x^2)


-- swap :: (a, b) -> (b, a)
swap =
   foldFunc [CT_Pair CT_Void CT_Void, CT_Pair CT_Void CT_Void] $
     \ [CO_Pair a b] -> return $ CO_Pair b a


-- getYear :: IO Int
getYear = CO_IO CT_Int $
            do clock_time <- getClockTime
               calendar_time <- toCalendarTime clock_time
               return (CO_Int (ctYear calendar_time))


-- hello :: IO ()
hello = CO_IO CT_Void $ do putStrLn "Hello, world!"
                           return CO_Void


-- map :: (a -> b) -> [a] -> [b]
map_ =
 foldFunc [CT_Func CT_Void CT_Void, CT_List CT_Void, CT_List CT_Void] $
     \ [CO_Func _ _ f, l] -> map' f l
    where
       map' :: (CommonObject -> IO CommonObject) -> CommonObject
```

```
            -> IO CommonObject
        map' _ CO_EmptyList      = return CO_EmptyList
        map' f (CO_List h t)     = do h' <- f h
                                      t' <- map' f t
                                      return $ CO_List h' t'


-- C :: Int -> Int -> Int
-- C(n, r) = n! / (r! (n-r)!)
-- computes the number of r-combinations of a set of n elements
-- using the factorial function defined in module V1
c rt =
   foldFunc [CT_Int, CT_Int, CT_Int] $
     \ [CO_Int n, CO_Int r] ->
         do -- obtain the factorial function from module V1
            fact     <- sym rt "V1" "factorial"
            -- evaluate the individual terms
            (CO_Int n_factorial)   <- apply_proc fact [CO_Int n]
            (CO_Int r_factorial)   <- apply_proc fact [CO_Int r]
            (CO_Int n_r_factorial) <- apply_proc fact [CO_Int (n-r)]
            return $ CO_Int (n_factorial `div`
                             (r_factorial * n_r_factorial))
```

Appendix 8. Sample V Module

```
module V1(
  -- exported symbols
  n, s, p, l, plusOne, swap, hello, timesFour, doubleAll,
  factorial, map
) where

-- imported modules
import C1;

-- some primitive data
n := 24;
s := "less is more";

-- a pair and a list
p := (0, 10);
l := ["a", "b", "c", "d", "e"];

-- a simple function adding one to its argument
func plusOne(x:Int):Int {
  return x+1
};

-- a function that swaps the components of a pair
func swap(a:(Void, Void)):(Void, Void) {
  return (right(a), left(a))
};
```

```
-- the canonical "Hello world!" function
func hello():Void {
  printLn("Hello, world!")
};

-- the map :: (a -> b) -> a -> [b] function
func map(f:Void, l:[Void]):[Void] {
  if l = []
    then return []
    else return (f(head(l)) : map(f, tail(l)))
};

-- the canonical factorial function
func factorial(n:Int):Int {
  -- the "if" is the C-style conditional -
  -- it evaluates the then-clause if the condition is non-zero
  if n
    then return n * factorial(n-1)
    else return 1
};

-- a function that multiplies its argument by four by applying
-- the double() function from moudule C1 twice
func timesFour(x:Int):Int {
   return C1.double(C1.double(x))
};
```

```
-- a function that doubles every components in a list using
-- the map() and double() functions of module C1
func doubleAll(x:[Int]):[Int] {
  return C1.map(C1.double, x)
}
```